US005491760A

United States Patent [19]

Withgott et al.

[11] **Patent Number: 5,491,760**

[45] **Date of Patent: Feb. 13, 1996**

[54] **METHOD AND APPARATUS FOR SUMMARIZING A DOCUMENT WITHOUT DOCUMENT IMAGE DECODING**

[75] Inventors: **M. Margaret Withgott**, Los Altos; **Steven C. Bagley**; **Dan S. Bloomberg**, both of Palo Alto; **Per-Kristian Halvorsen**, Los Altos, all of Calif.; **Daniel P. Huttenlocher**, Ithaca, N.Y.; **Todd A. Cass**, Cambridge, Mass.; **Ronald M. Kaplan**, Palo Alto; **Ramana R. Rao**, San Francisco, both of Calif.

[73] Assignee: **Xerox Corporation**, Stamford, Conn.

[21] Appl. No.: **240,284**

[22] Filed: **May 9, 1994**

Related U.S. Application Data

[63] Continuation of Ser. No. 794,543, Nov. 19, 1991, abandoned.

[51] **Int. Cl.^{6}** **G06K 9/46**

[52] **U.S. Cl.** **382/203**; 382/177; 382/229

[58] **Field of Search** 382/9, 55, 1, 28, 382/30, 25, 40, 177, 190, 114, 198, 199, 200, 203, 209, 206, 229, 257, 308; 364/419.03, 419.19

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,659,354	5/1972	Sutherland	35/35 A
4,654,873	3/1987	Fujisawa et al.	382/9
4,685,135	8/1987	Lin et al.	382/1
4,752,772	6/1988	Litt et al.	340/712
4,972,349	11/1990	Kleinberger	364/900
4,994,987	2/1991	Baldwin	364/518
5,048,109	9/1991	Bloomberg et al.	382/9
5,077,668	12/1991	Doi	364/419
5,131,049	7/1992	Bloomberg et al.	382/55
5,181,255	1/1993	Bloomberg et al.	382/55
5,202,933	4/1993	Bloomberg et al.	382/55
5,216,725	6/1993	McCubbrey	382/9
5,325,444	6/1994	Cass et al.	382/40
5,384,863	1/1995	Huttenlocher et al.	382/40

FOREIGN PATENT DOCUMENTS

0361464	4/1990	European Pat. Off.	15/401
59-135576	8/1984	Japan .	
02093866	4/1990	Japan .	

OTHER PUBLICATIONS

"A Business Intelligence System" by H. P. Luhn, IBM Journal, Oct. 1958.

D. S. Bloomberg "Multiresolution morphological approach to document image analysis" First International Conference on Document Analysis and Recognition ICDAR 91, 30 Sep.–2 Oct. 1991, St. Malo, France pp. 963–971.

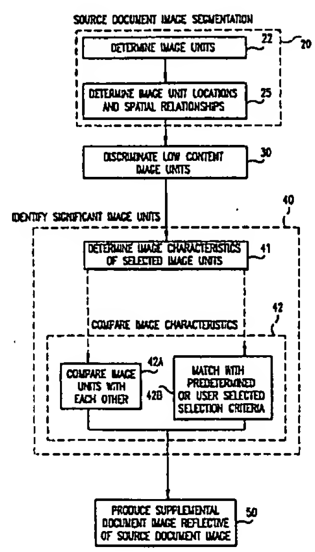
M. Hase et al. "A method for extracting marked regions from document images" Proc. 8th International Conference On Pattern Recognition, 27–31 Oct. 1986, Paris France, pp. 780–782.

Primary Examiner—Leo Boudreau
Assistant Examiner—Phuoc Tran
Attorney, Agent, or Firm—Oliff & Berridge

[57] **ABSTRACT**

A method and apparatus for excerpting and summarizing an undecoded document image, without first converting the document image to optical character codes such as ASCII text, identifies significant words, phrases and graphics in the document image using automatic or interactive morphological image recognition techniques, document summaries or indices are produced based on the identified significant portions of the document image. The disclosed method is particularly adept for improvement of reading machines for the blind.

20 Claims, 39 Drawing Sheets



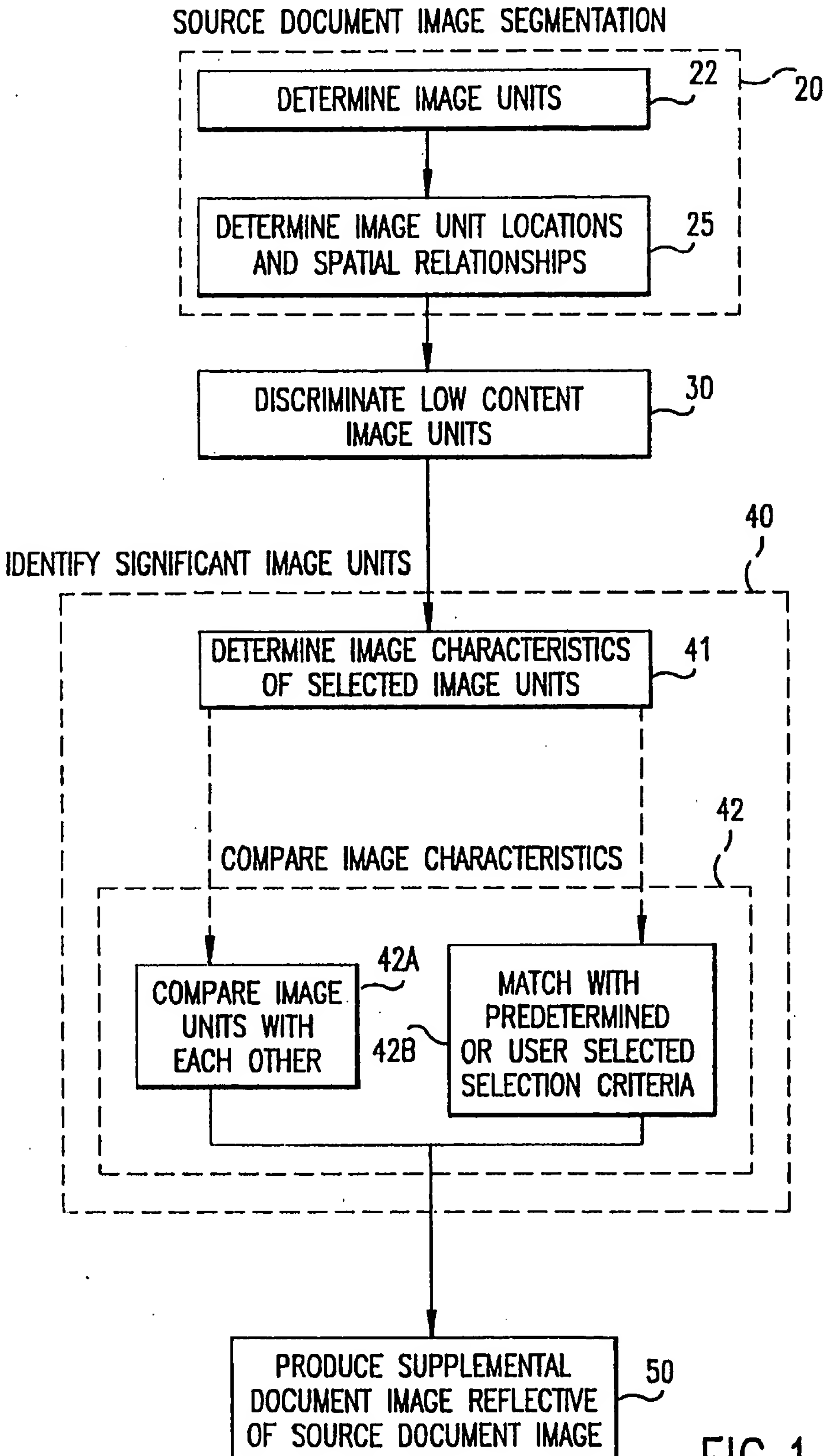

SOURCE DOCUMENT IMAGE SEGMENTATION
DETERMINE IMAGE UNITS
22
20
DETERMINE IMAGE UNIT LOCATIONS AND SPATIAL RELATIONSHIPS
25
DISCRIMINATE LOW CONTENT IMAGE UNITS
30
IDENTIFY SIGNIFICANT IMAGE UNITS
40
DETERMINE IMAGE CHARACTERISTICS OF SELECTED IMAGE UNITS
41
COMPARE IMAGE CHARACTERISTICS
42
COMPARE IMAGE UNITS WITH EACH OTHER
42A
42B
MATCH WITH PREDETERMINED OR USER SELECTED SELECTION CRITERIA
PRODUCE SUPPLEMENTAL DOCUMENT IMAGE REFLECTIVE OF SOURCE DOCUMENT IMAGE
50

FIG.1

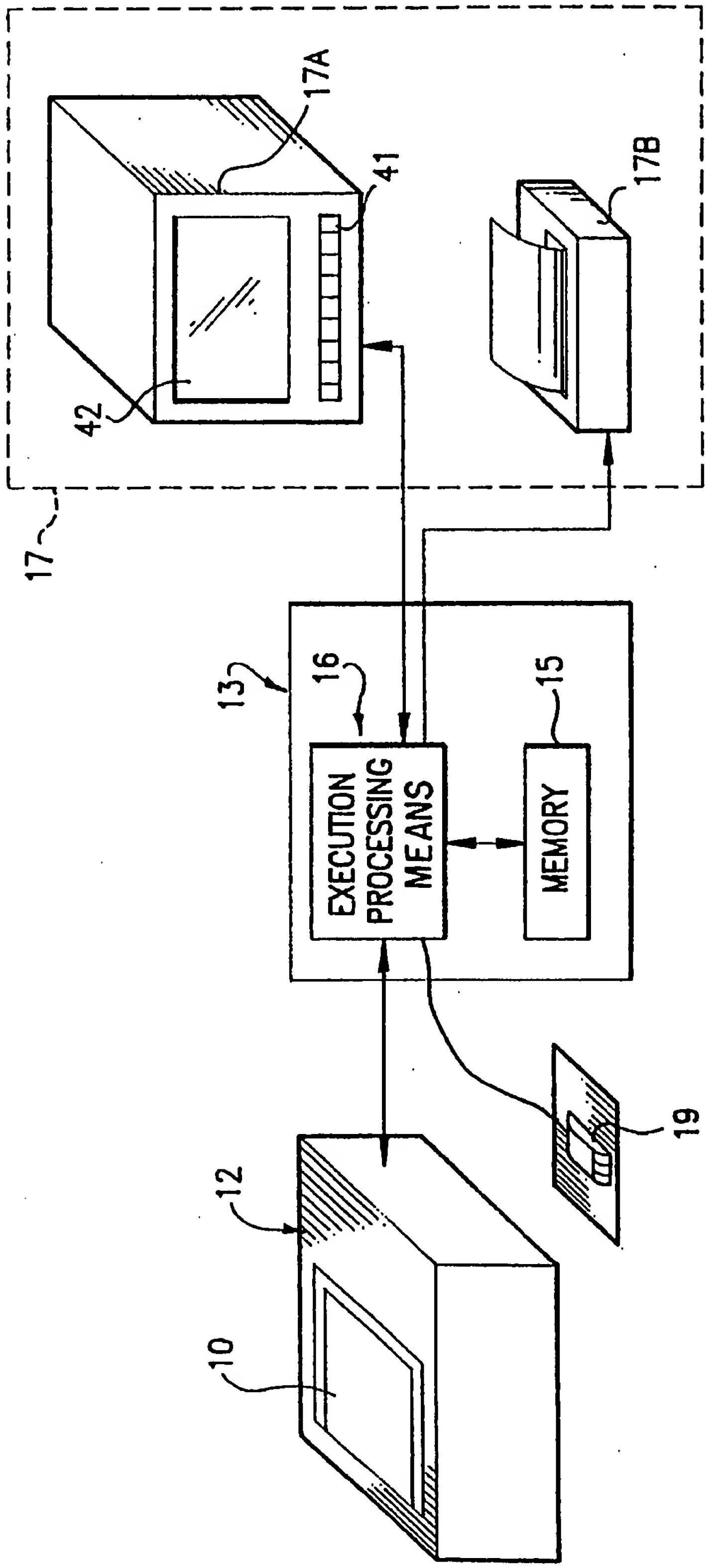
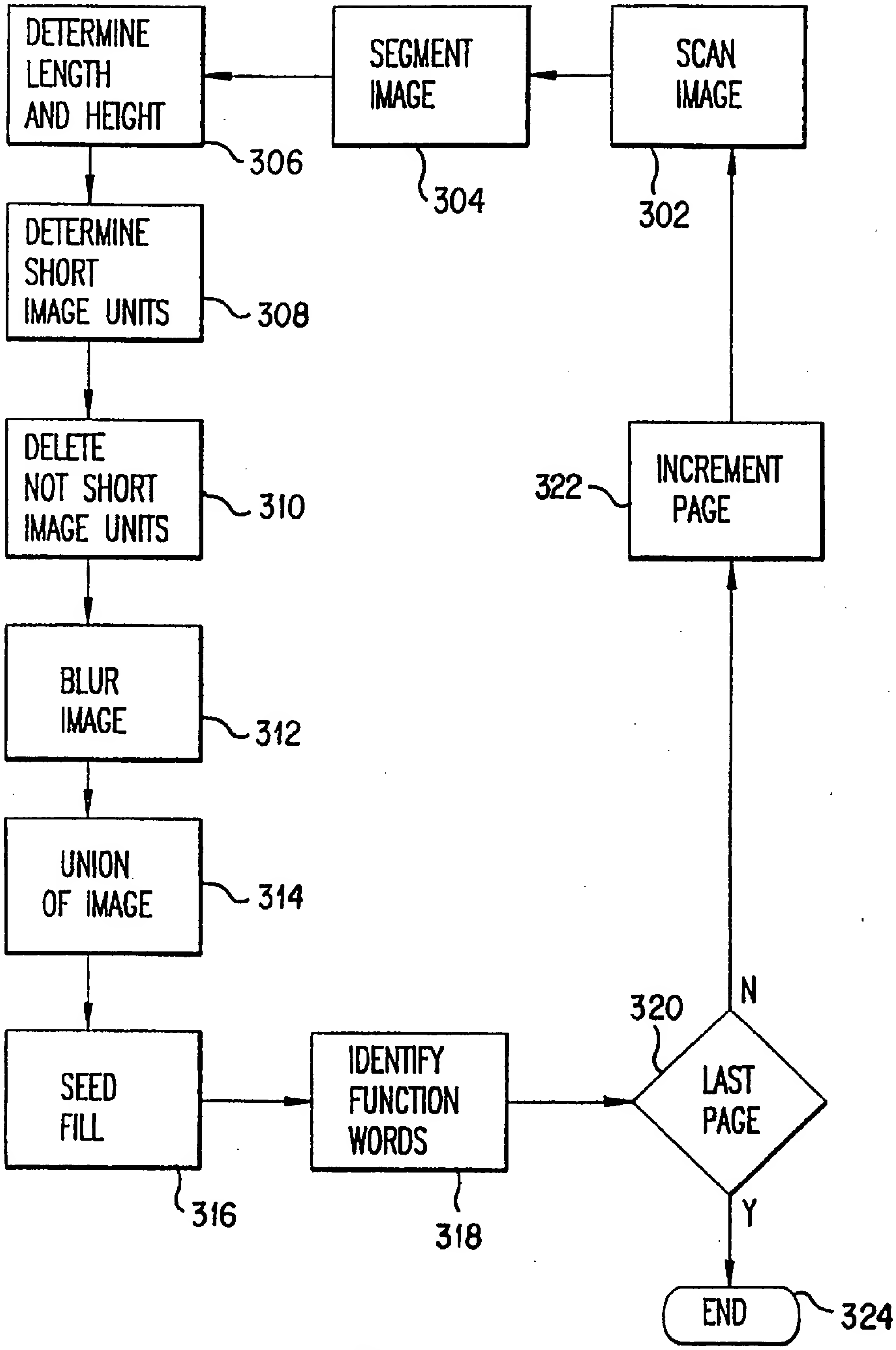

17A
41
17B
42
17
13
16
15
EXECUTION PROCESSING MEANS
MEMORY
12
19
10

FIG.2



DETERMINE LENGTH AND HEIGHT
306
SEGMENT IMAGE
304
SCAN IMAGE
302
DETERMINE SHORT IMAGE UNITS
308
DELETE NOT SHORT IMAGE UNITS
310
322
INCREMENT PAGE
BLUR IMAGE
312
UNION OF IMAGE
314
320
N
SEED FILL
316
IDENTIFY FUNCTION WORDS
318
LAST PAGE
Y
END
324

FIG. 3

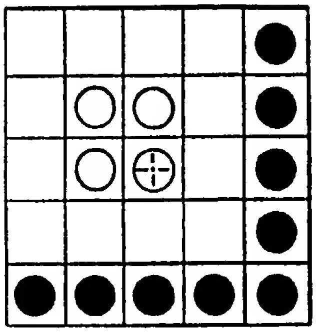


FIG. 4E

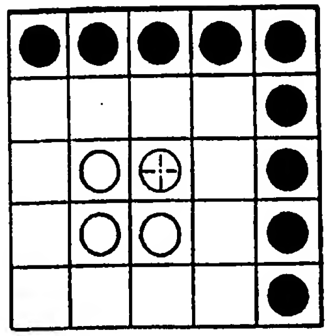


FIG. 4F

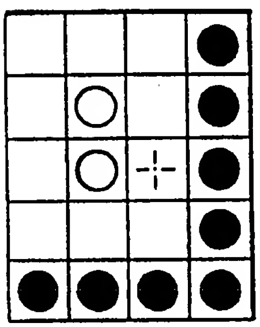


FIG. 4C

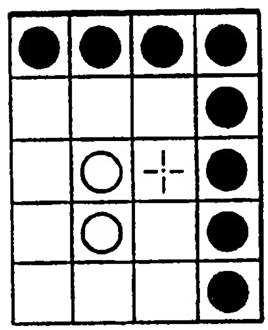


FIG. 4D

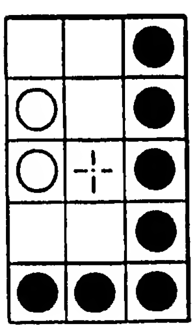


FIG. 4A

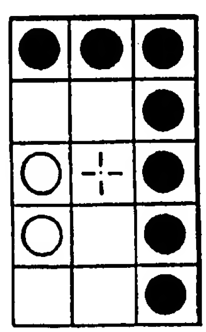


FIG. 4B

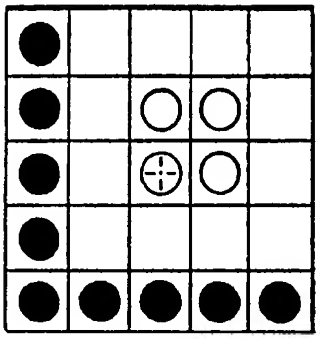


FIG. 5E

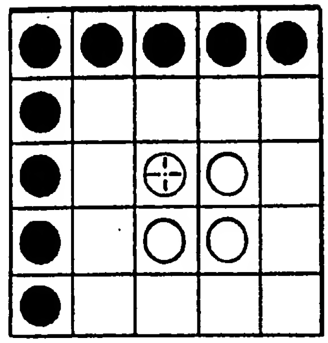


FIG. 5F

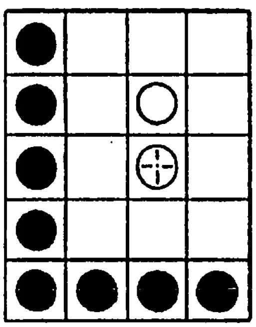


FIG. 5C

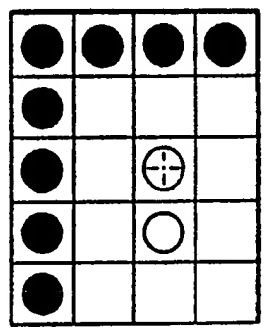


FIG. 5D

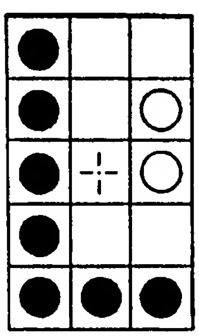


FIG. 5A

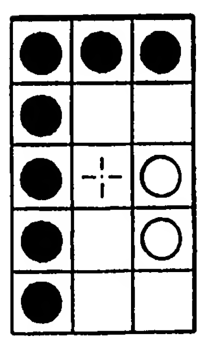


FIG. 5B

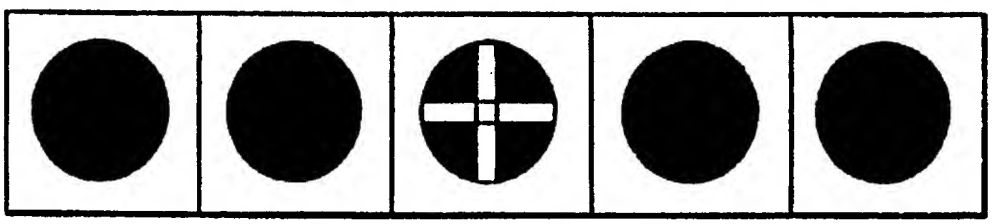


FIG. 6

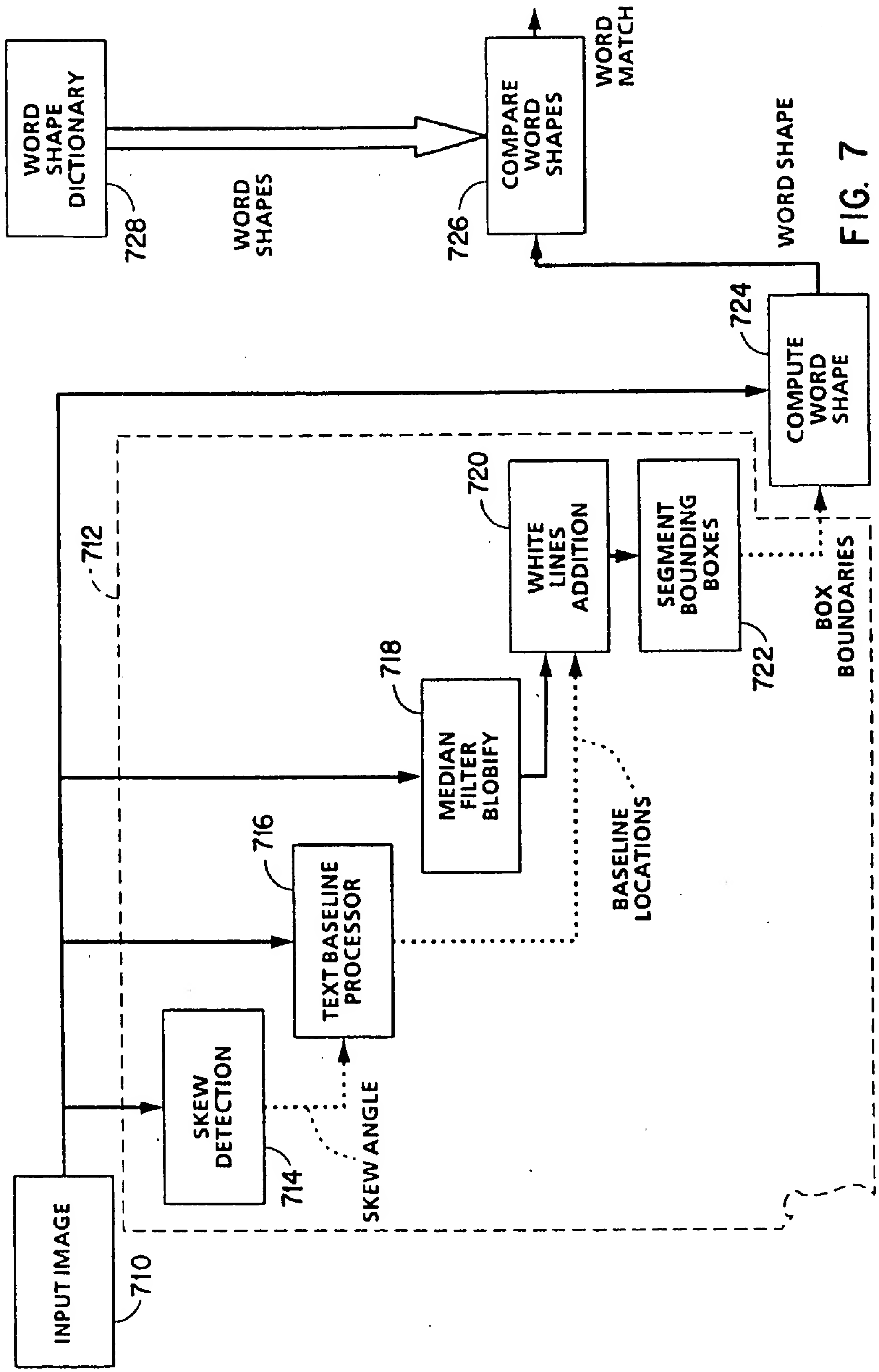

WORD SHAPE DICTIONARY
728
WORD SHAPES
COMPARE WORD SHAPES
726
WORD MATCH
WORD SHAPE
COMPUTE WORD SHAPE
724
712
WHITE LINES ADDITION
720
SEGMENT BOUNDING BOXES
722
BOX BOUNDARIES
MEDIAN FILTER BLOBIFY
718
BASELINE LOCATIONS
TEXT BASELINE PROCESSOR
716
SKEW DETECTION
714
SKEW ANGLE
INPUT IMAGE
710

FIG. 7

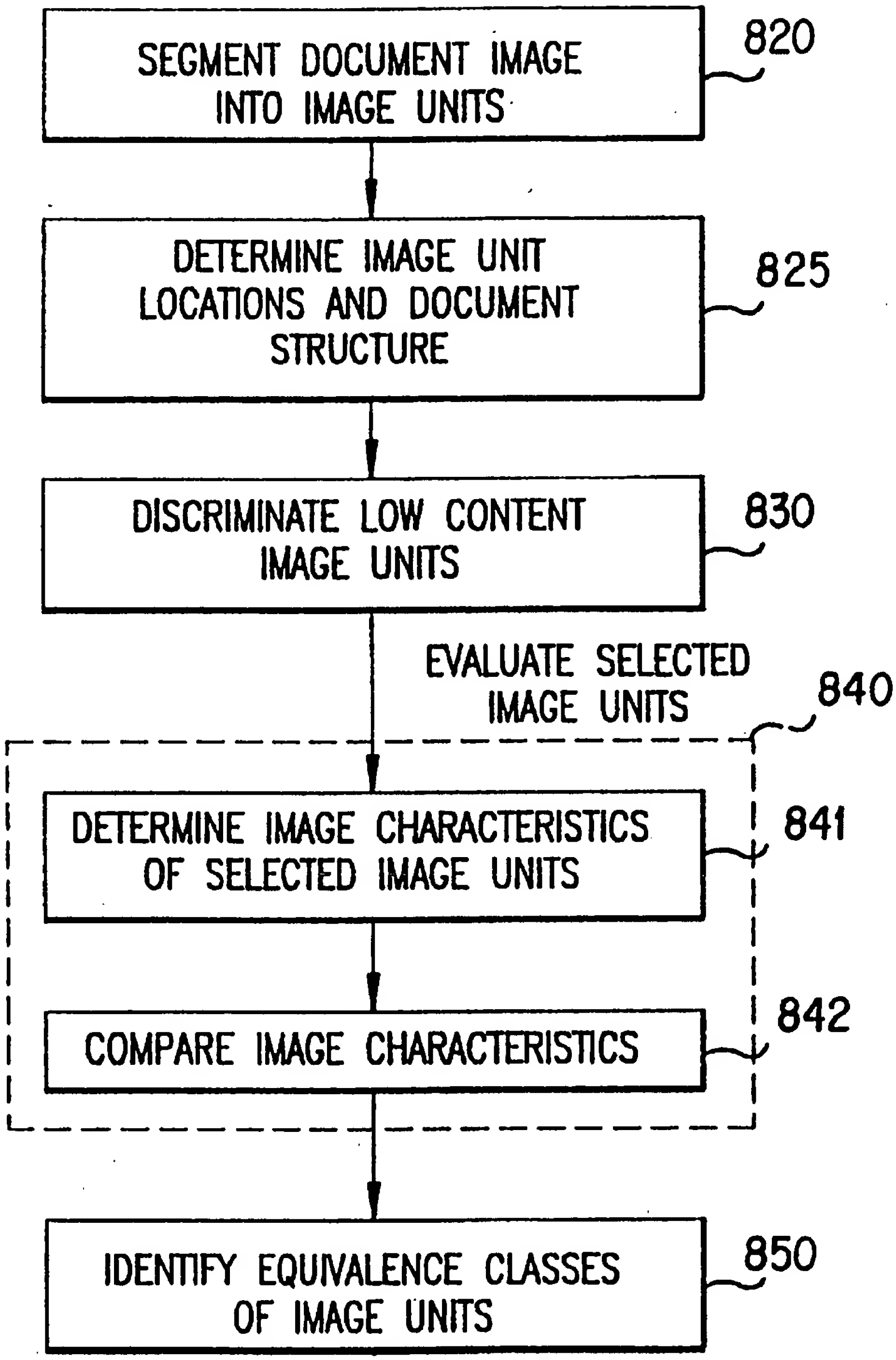

SEGMENT DOCUMENT IMAGE INTO IMAGE UNITS
820
DETERMINE IMAGE UNIT LOCATIONS AND DOCUMENT STRUCTURE
825
DISCRIMINATE LOW CONTENT IMAGE UNITS
830
EVALUATE SELECTED IMAGE UNITS
840
DETERMINE IMAGE CHARACTERISTICS OF SELECTED IMAGE UNITS
841
COMPARE IMAGE CHARACTERISTICS
842
IDENTIFY EQUIVALENCE CLASSES OF IMAGE UNITS
850

FIG. 8

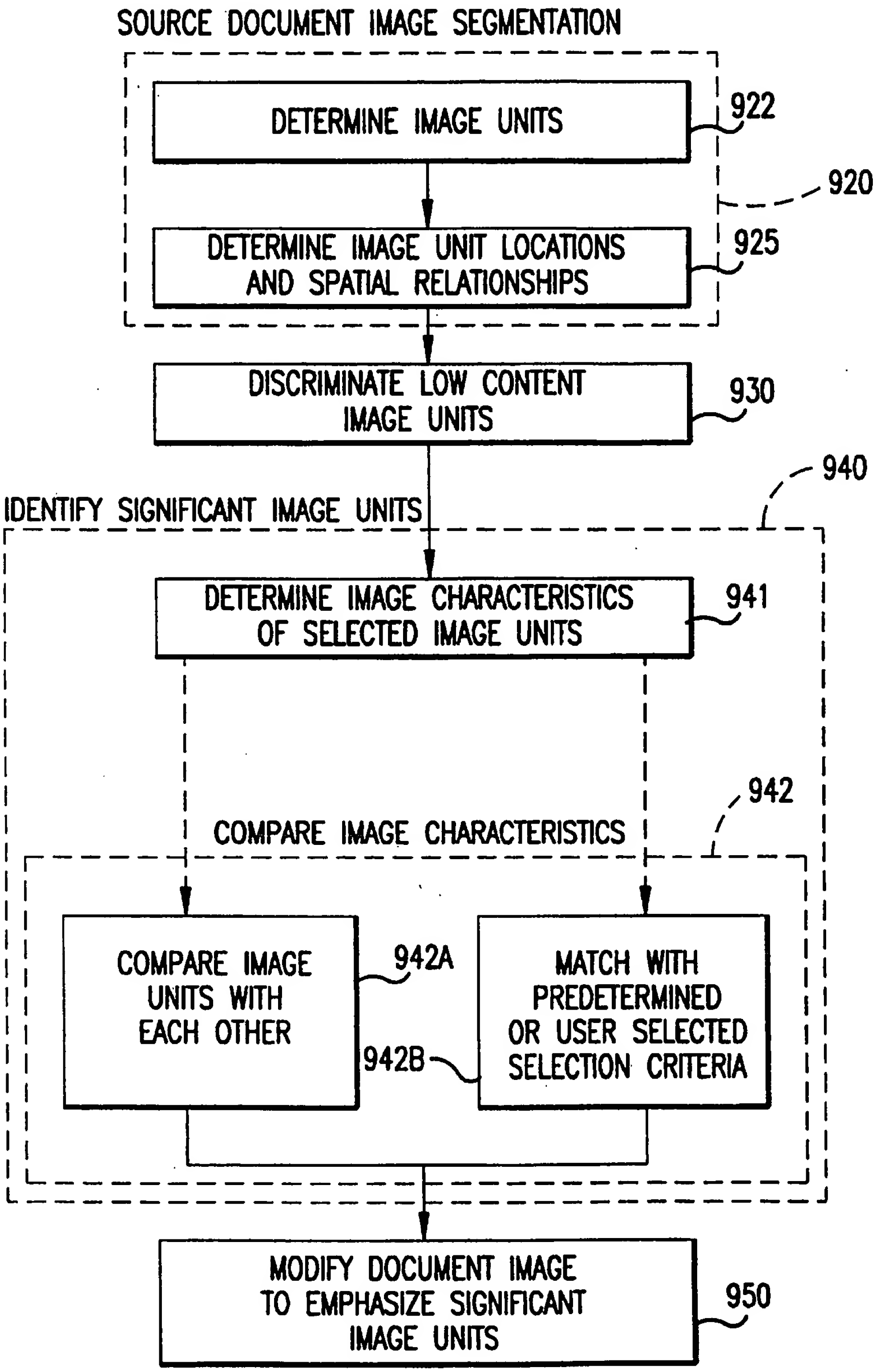

SOURCE DOCUMENT IMAGE SEGMENTATION
DETERMINE IMAGE UNITS
922
920
DETERMINE IMAGE UNIT LOCATIONS AND SPATIAL RELATIONSHIPS
925
DISCRIMINATE LOW CONTENT IMAGE UNITS
930
940
IDENTIFY SIGNIFICANT IMAGE UNITS
DETERMINE IMAGE CHARACTERISTICS OF SELECTED IMAGE UNITS
941
942
COMPARE IMAGE CHARACTERISTICS
COMPARE IMAGE UNITS WITH EACH OTHER
942A
942B
MATCH WITH PREDETERMINED OR USER SELECTED SELECTION CRITERIA
MODIFY DOCUMENT IMAGE TO EMPHASIZE SIGNIFICANT IMAGE UNITS
950

FIG.9

A practitioner may not solicit professional employment from a prospective client with whom the practitioner has no family or prior professional relationship, by mail, in-person or otherwise, when a significant motive for the practitioner's doing so is the practitioner's pecuniary gain under circumstances evidencing undue influence, in-

FIG. 10

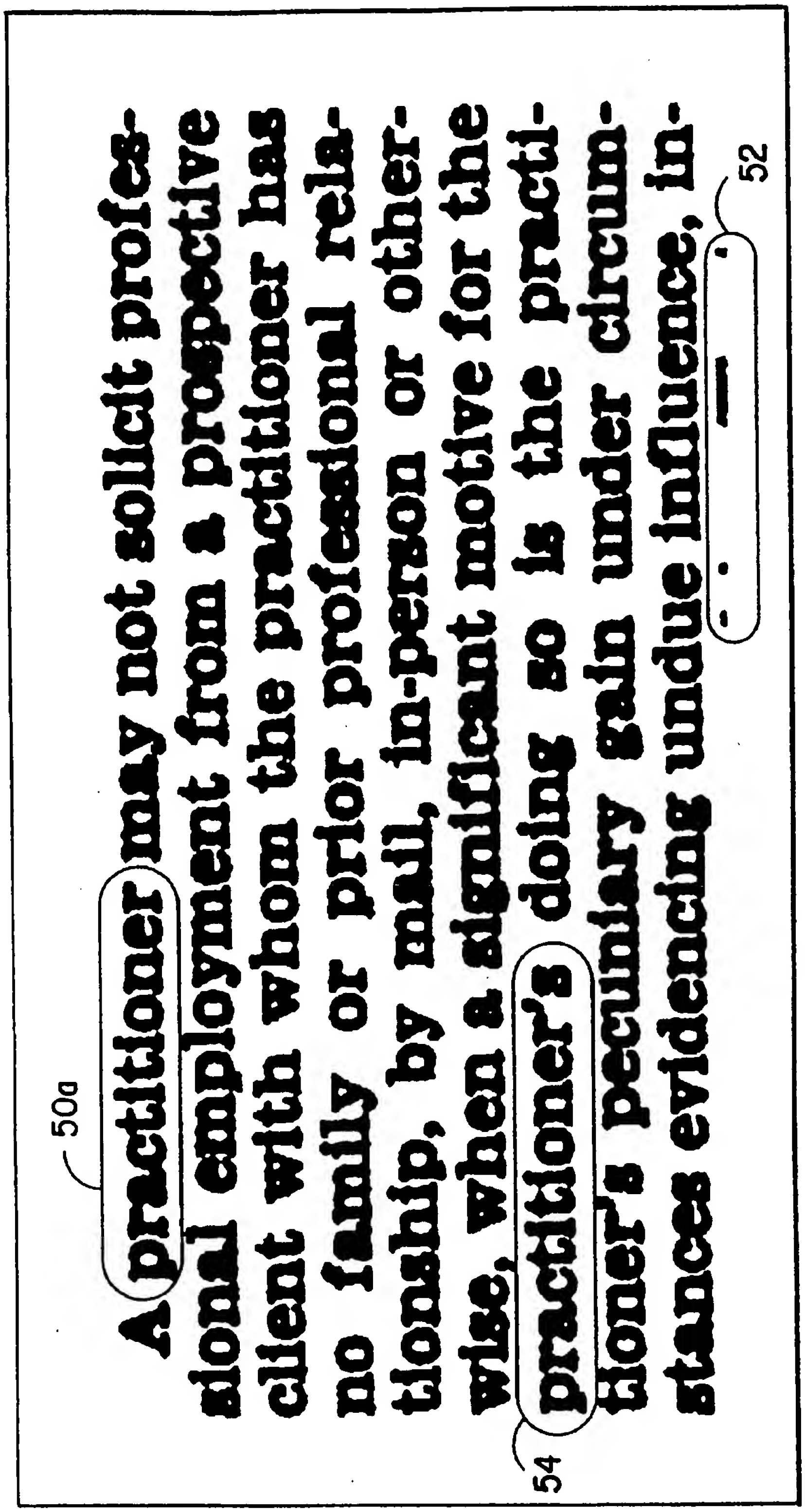

A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-
50a
52
54

FIG.11

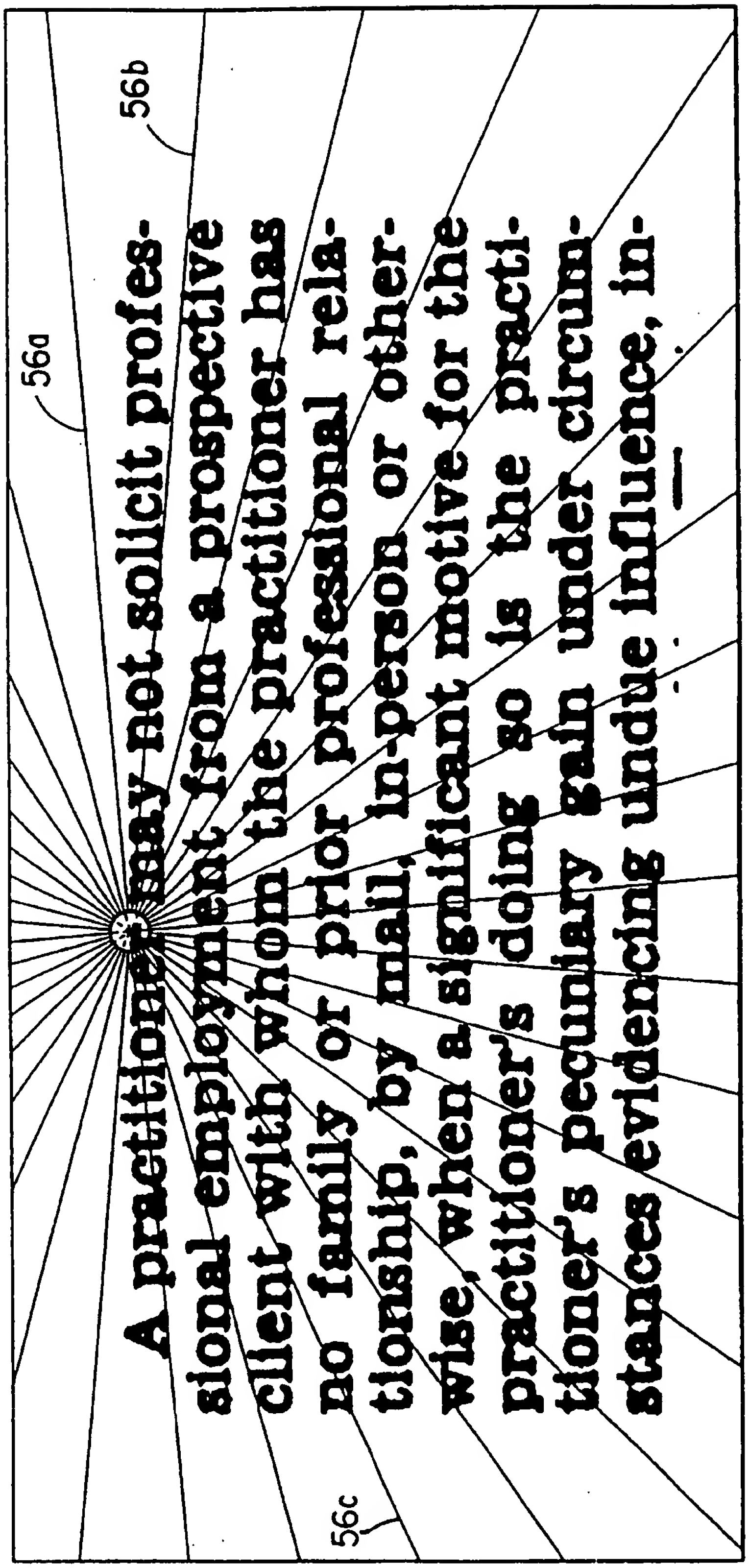

56a
56b
56c
A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-

FIG. 12A

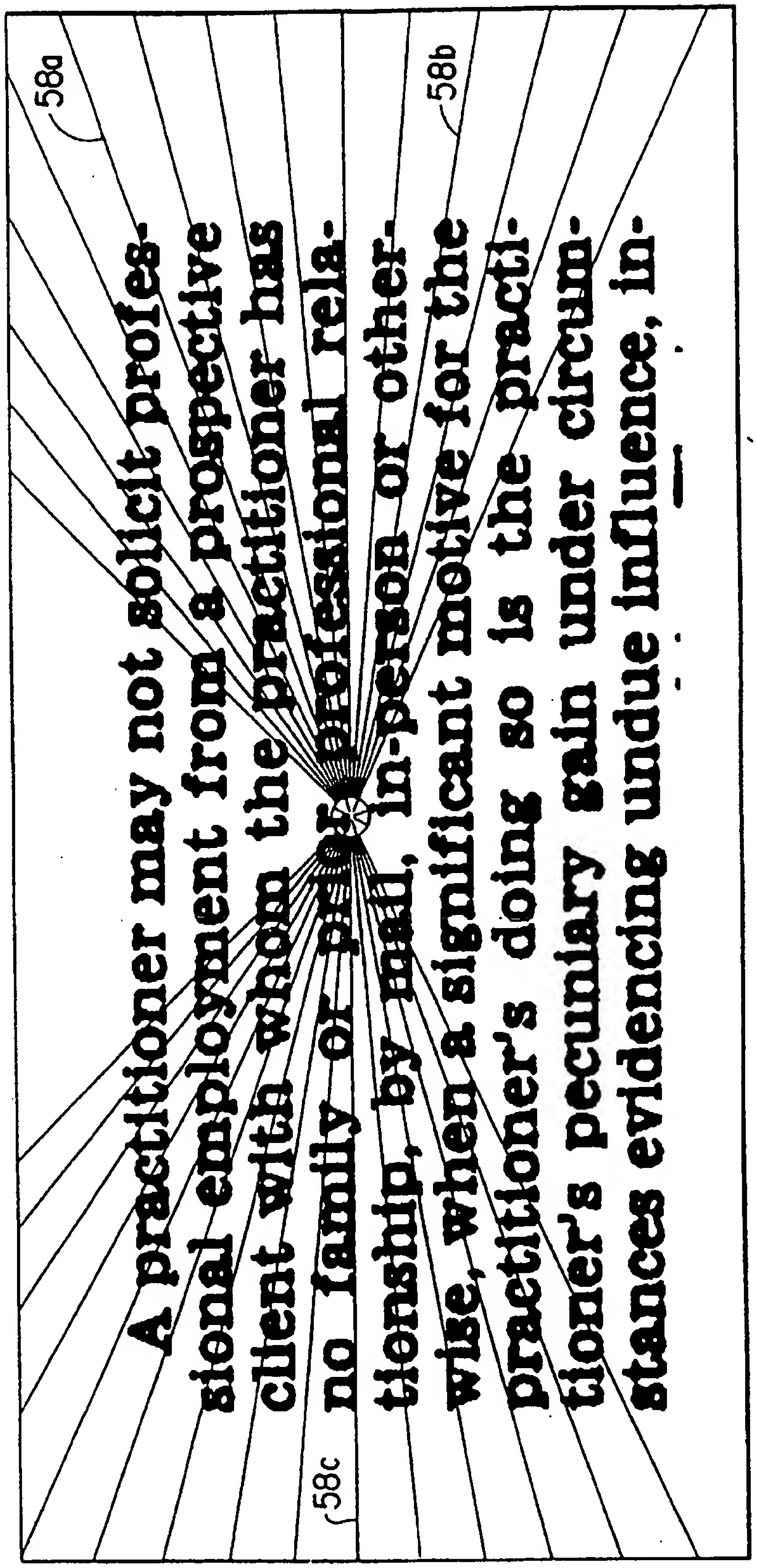

58a
58b
58c
A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-

FIG. 12B

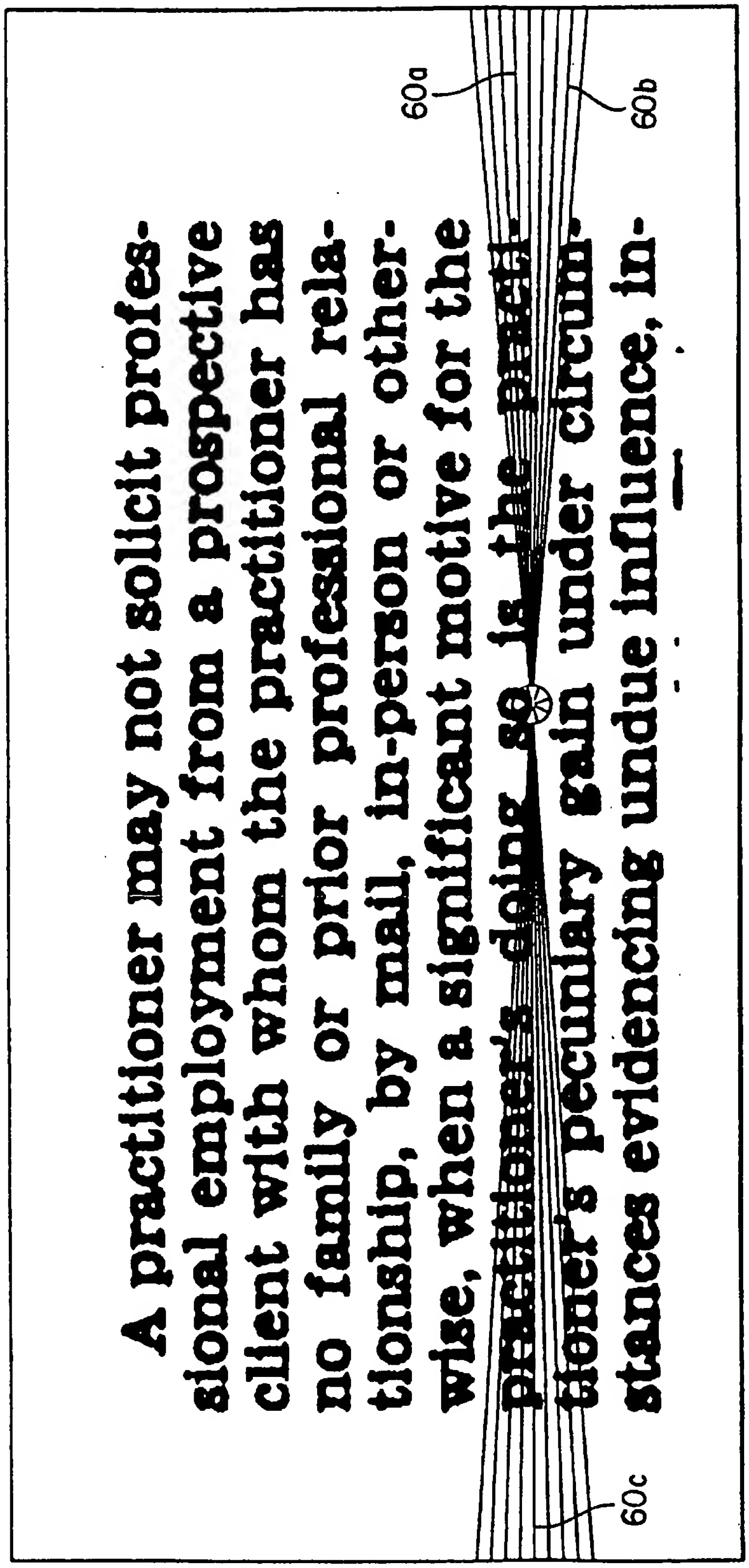

60a
60b
60c
A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-

FIG. 12C

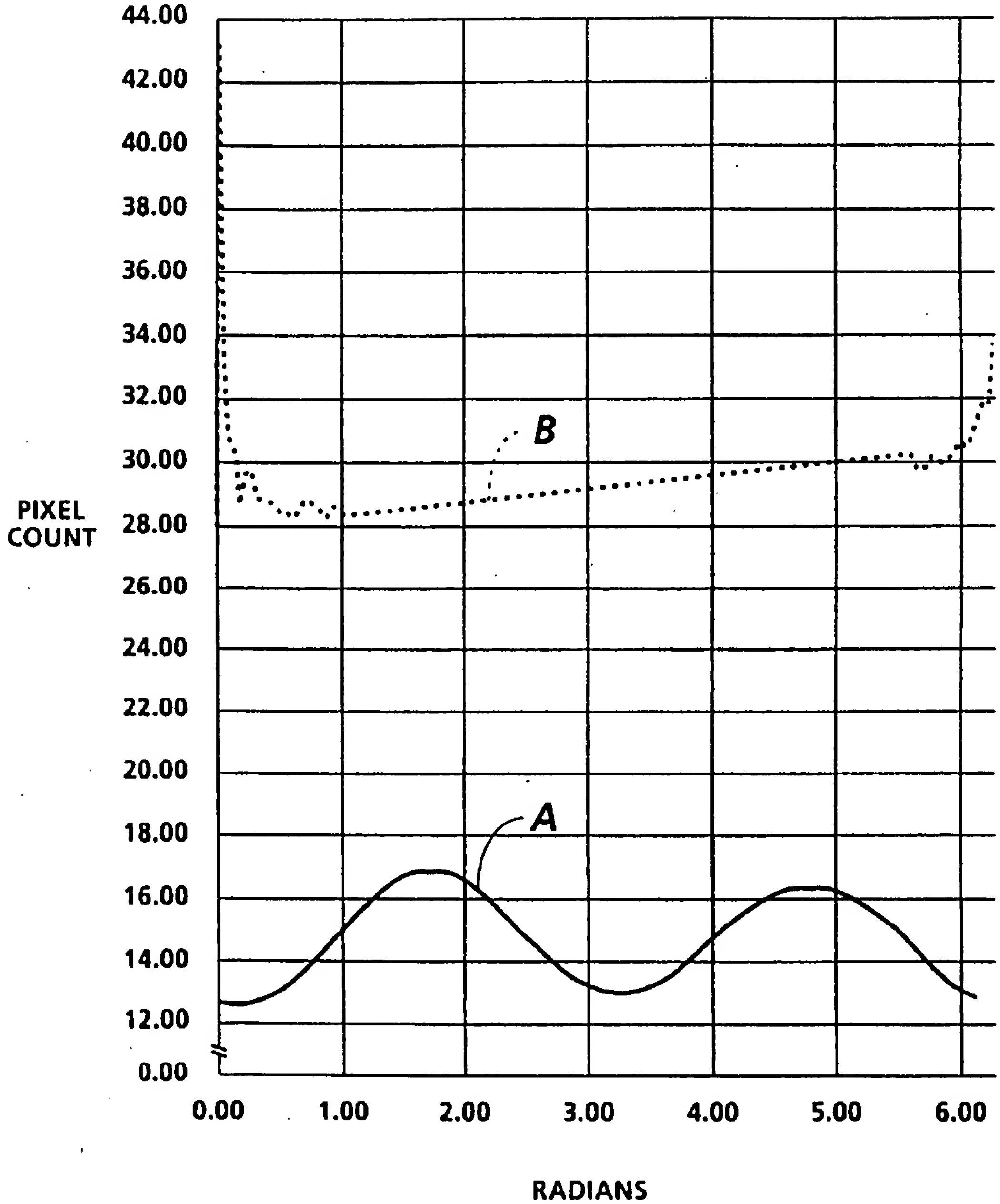

44.00
42.00
40.00
38.00
36.00
34.00
32.00
30.00
28.00
26.00
24.00
22.00
20.00
18.00
16.00
14.00
12.00
0.00
PIXEL COUNT
B
A
0.00
1.00
2.00
3.00
4.00
5.00
6.00
RADIANS

FIG. 12D

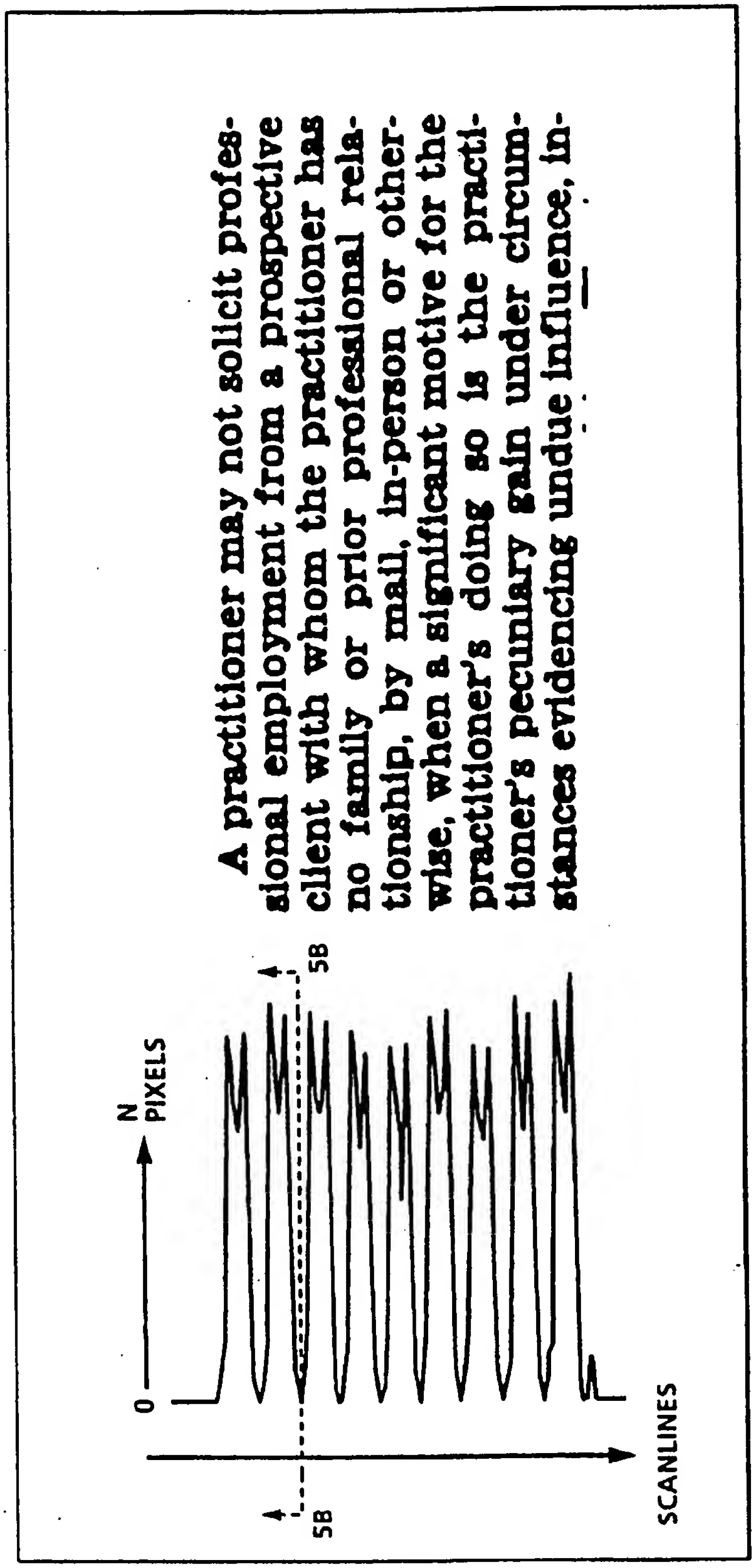

A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-
N PIXELS
0
SCANLINES
5B
5B

FIG. 13A

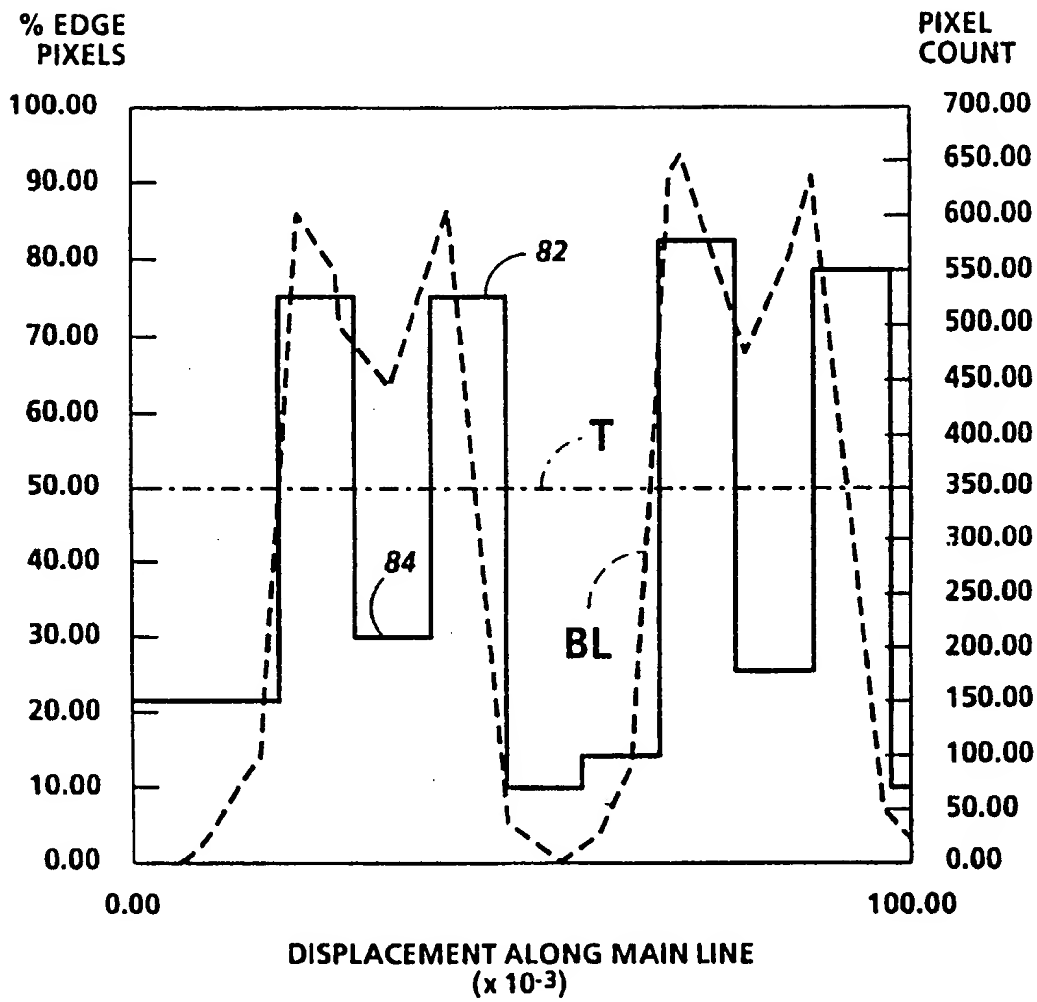

% EDGE PIXELS
PIXEL COUNT
100.00
90.00
80.00
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
700.00
650.00
600.00
550.00
500.00
450.00
400.00
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
82
84
T
BL
0.00
100.00
DISPLACEMENT ALONG MAIN LINE
(x 10-3)

FIG. 13 B

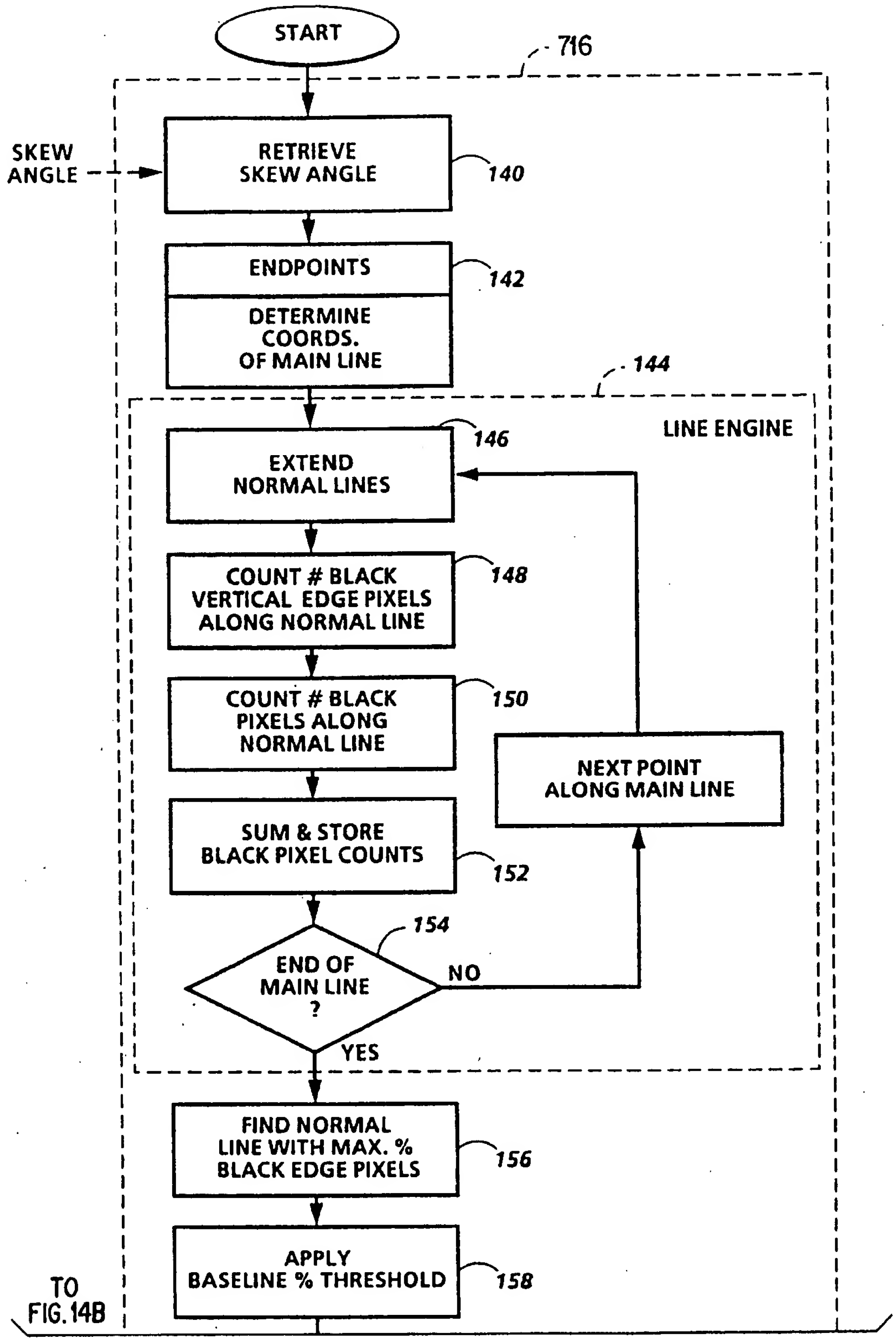

START
716
SKEW ANGLE
RETRIEVE SKEW ANGLE
140
ENDPOINTS
DETERMINE COORDS. OF MAIN LINE
142
144
LINE ENGINE
EXTEND NORMAL LINES
146
COUNT # BLACK VERTICAL EDGE PIXELS ALONG NORMAL LINE
148
COUNT # BLACK PIXELS ALONG NORMAL LINE
150
NEXT POINT ALONG MAIN LINE
SUM & STORE BLACK PIXEL COUNTS
152
154
END OF MAIN LINE ?
NO
YES
FIND NORMAL LINE WITH MAX. % BLACK EDGE PIXELS
156
APPLY BASELINE % THRESHOLD
158
TO FIG. 14B

FIG. 14A

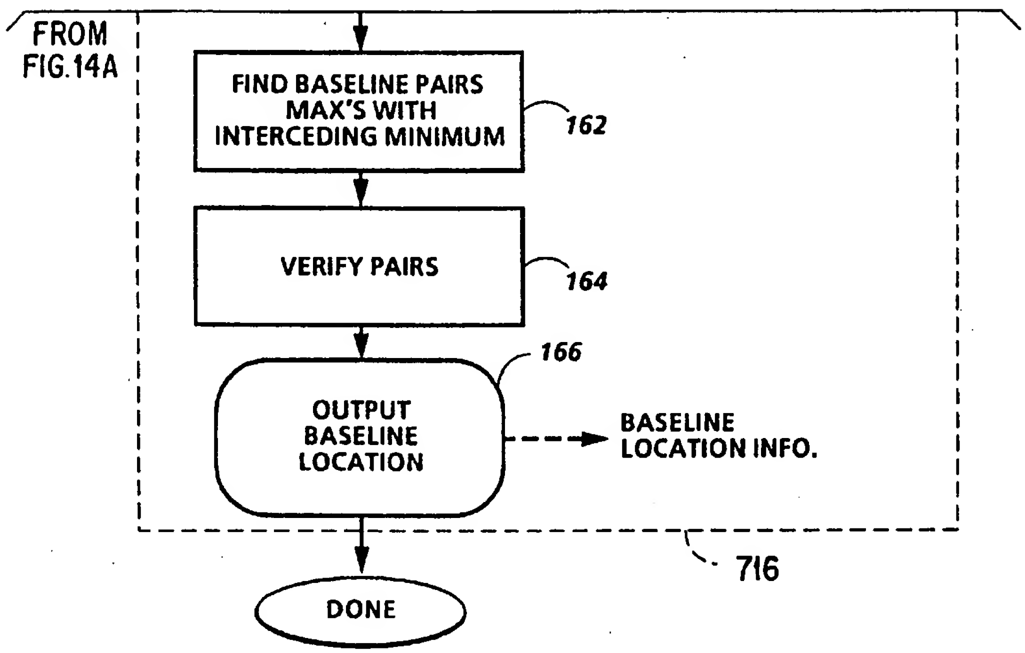

FROM
FIG.14A
FIND BASELINE PAIRS
MAX'S WITH
INTERCEDING MINIMUM
162
VERIFY PAIRS
164
166
OUTPUT
BASELINE
LOCATION
BASELINE
LOCATION INFO.
716
DONE

FIG. 14B

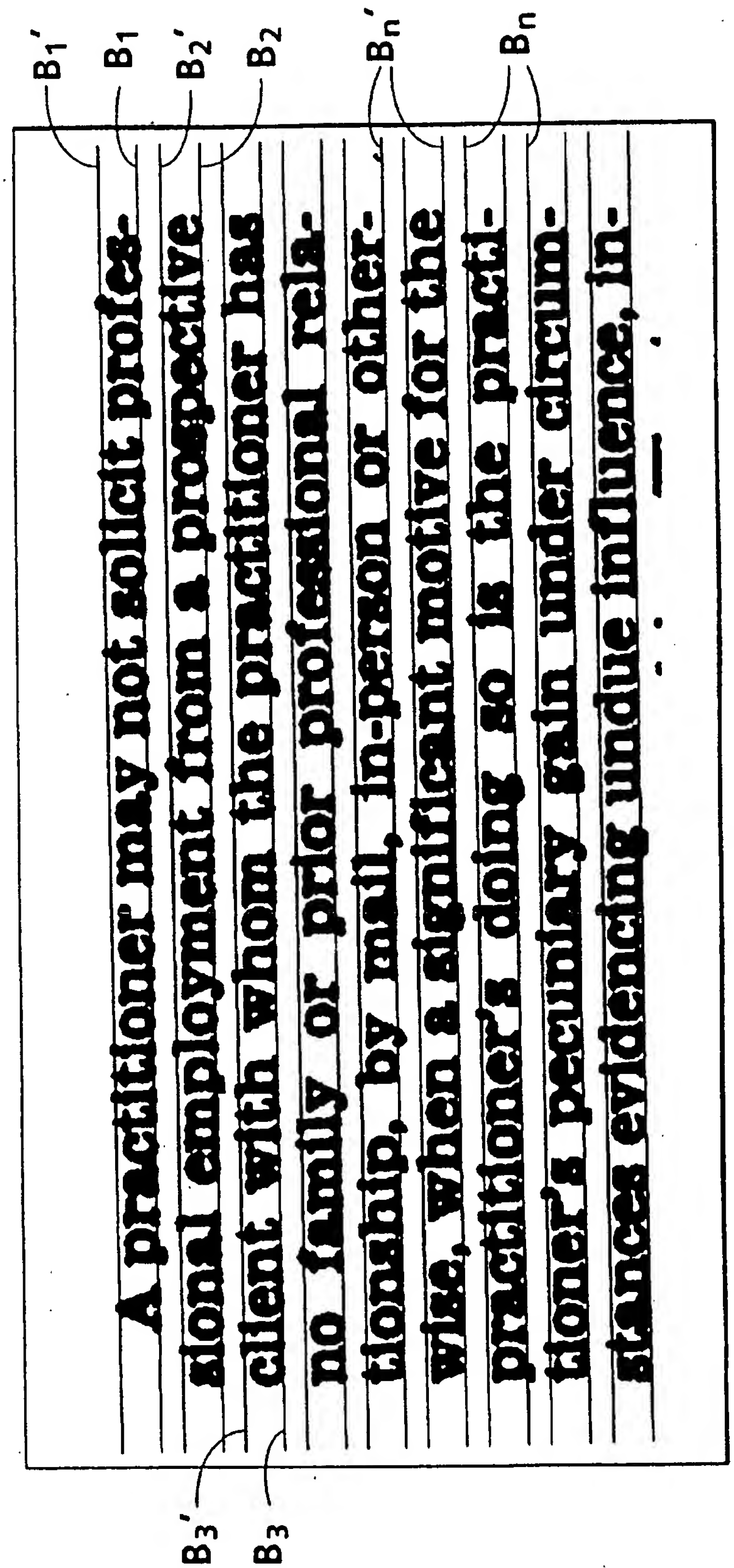

B_1'
B_1
B_2'
B_2
B_n'
B_n
A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-
B_3'
B_3

FIG. 15

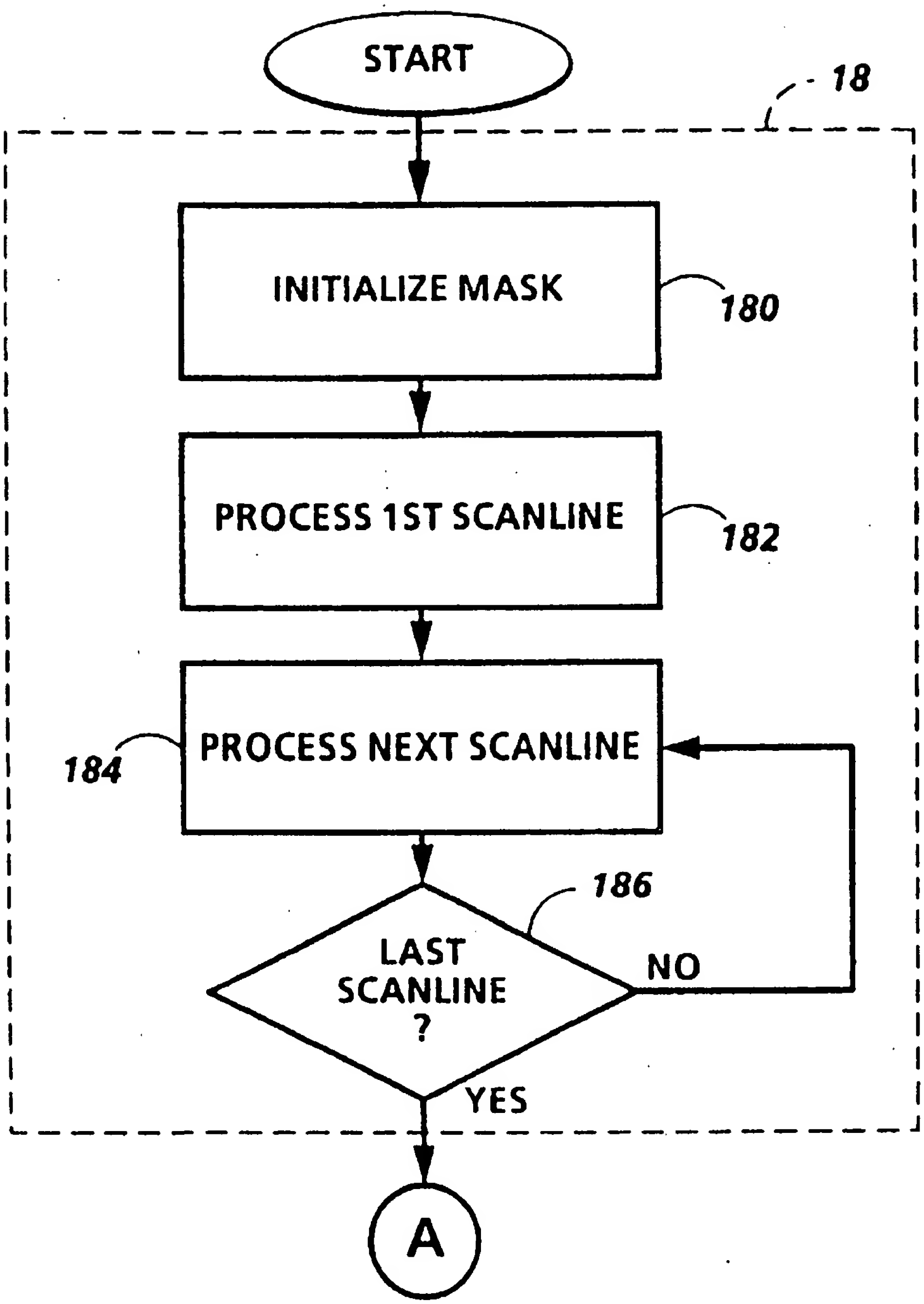

START
18
INITIALIZE MASK
180
PROCESS 1ST SCANLINE
182
PROCESS NEXT SCANLINE
184
186
LAST SCANLINE ?
NO
YES
A

FIG. 16

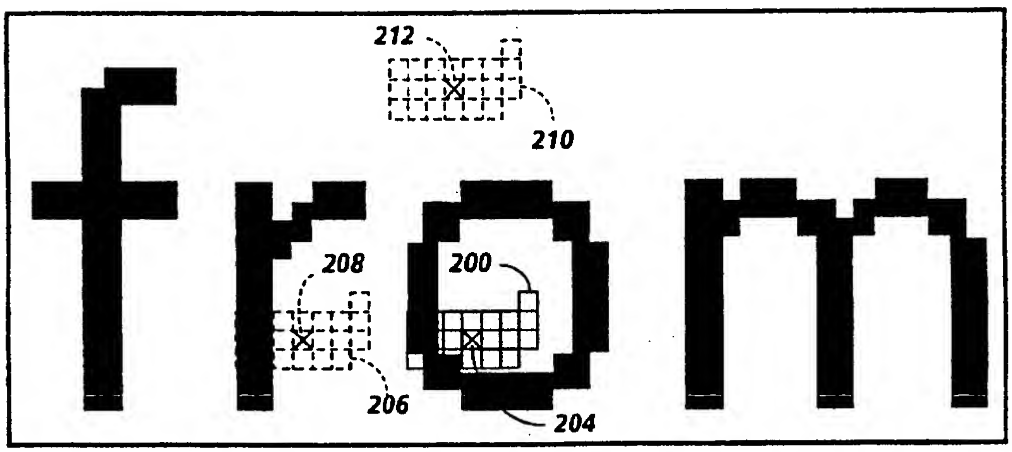


FIG. 17

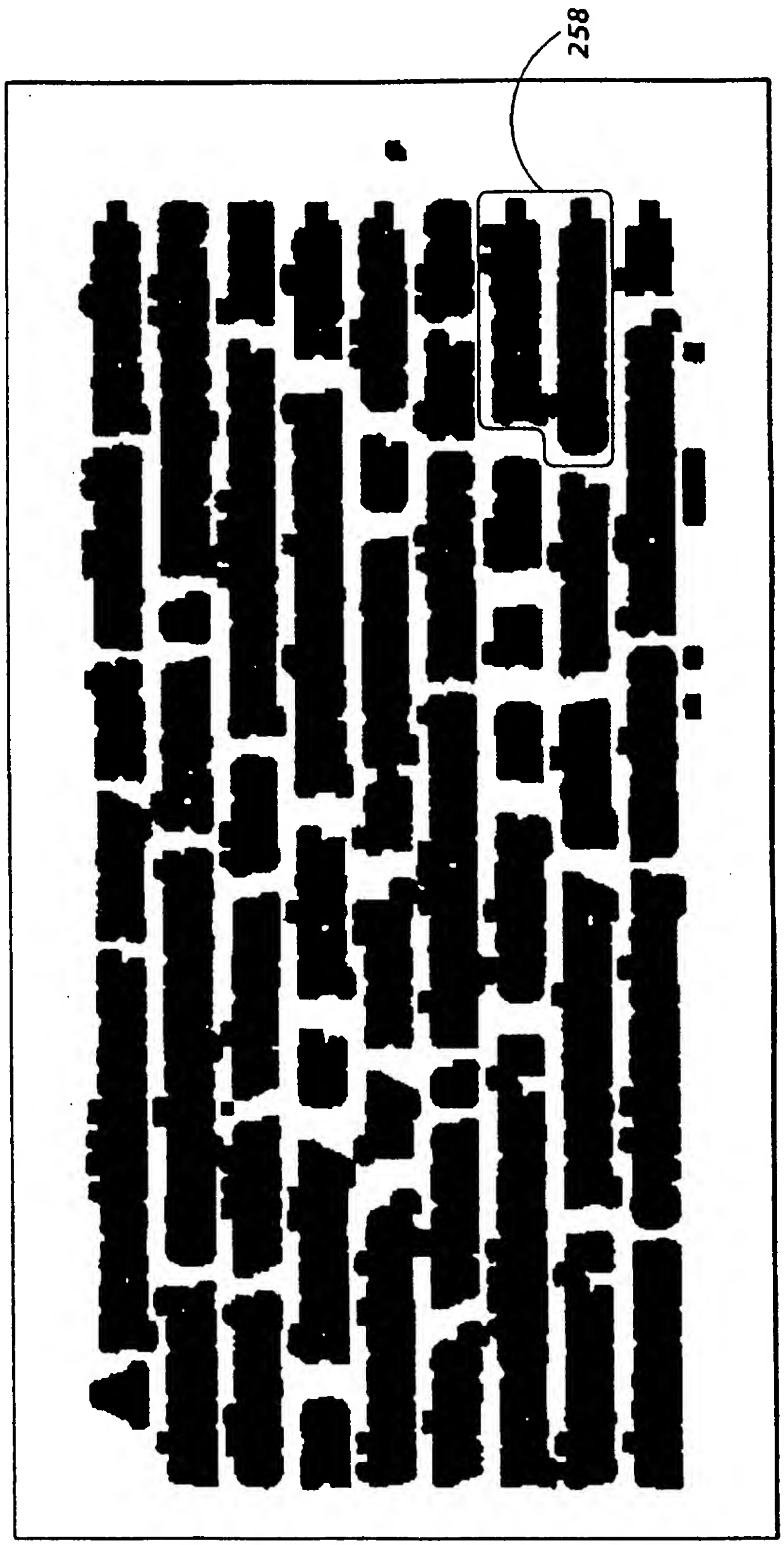


FIG. 18

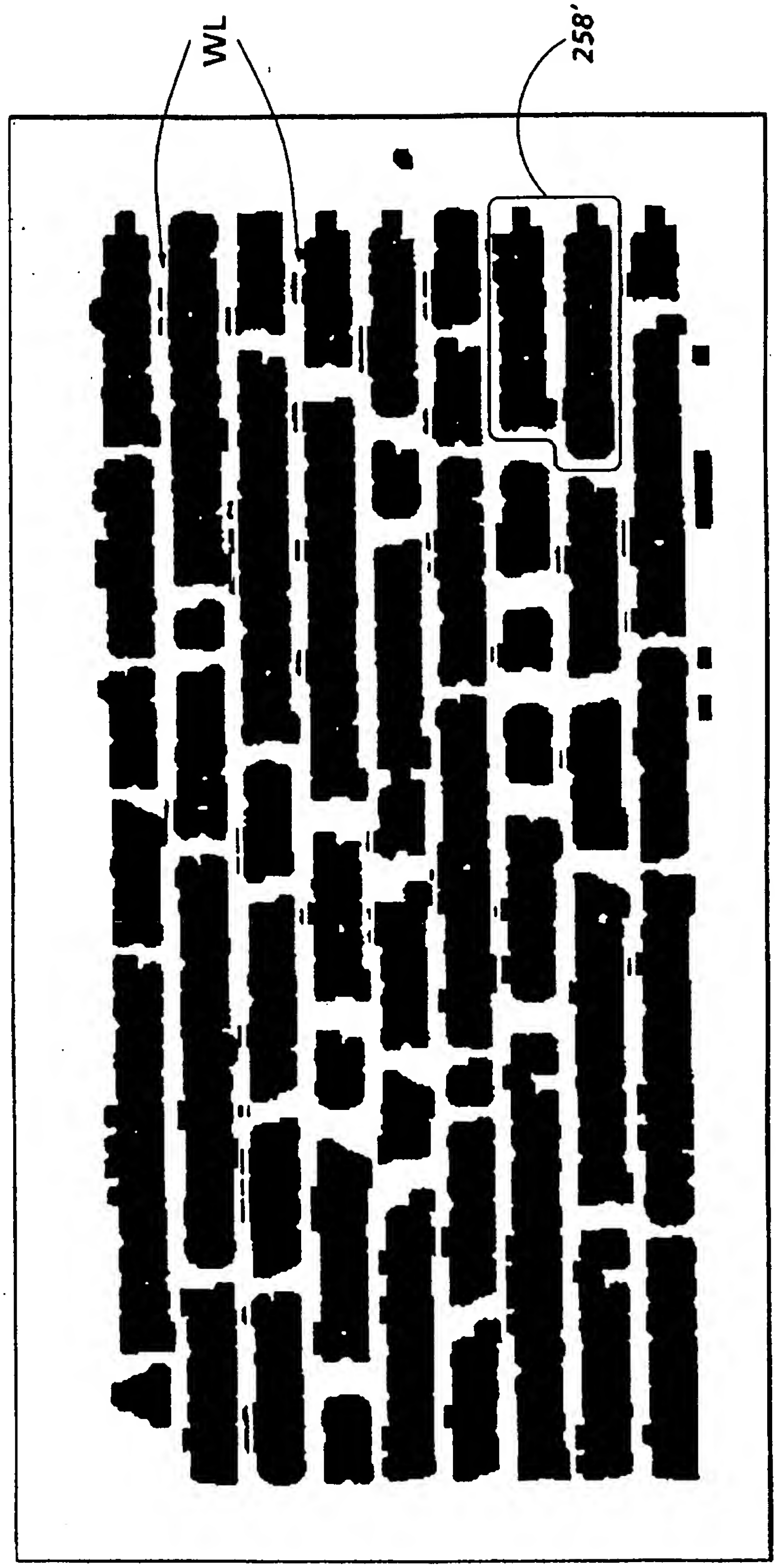


FIG. 19

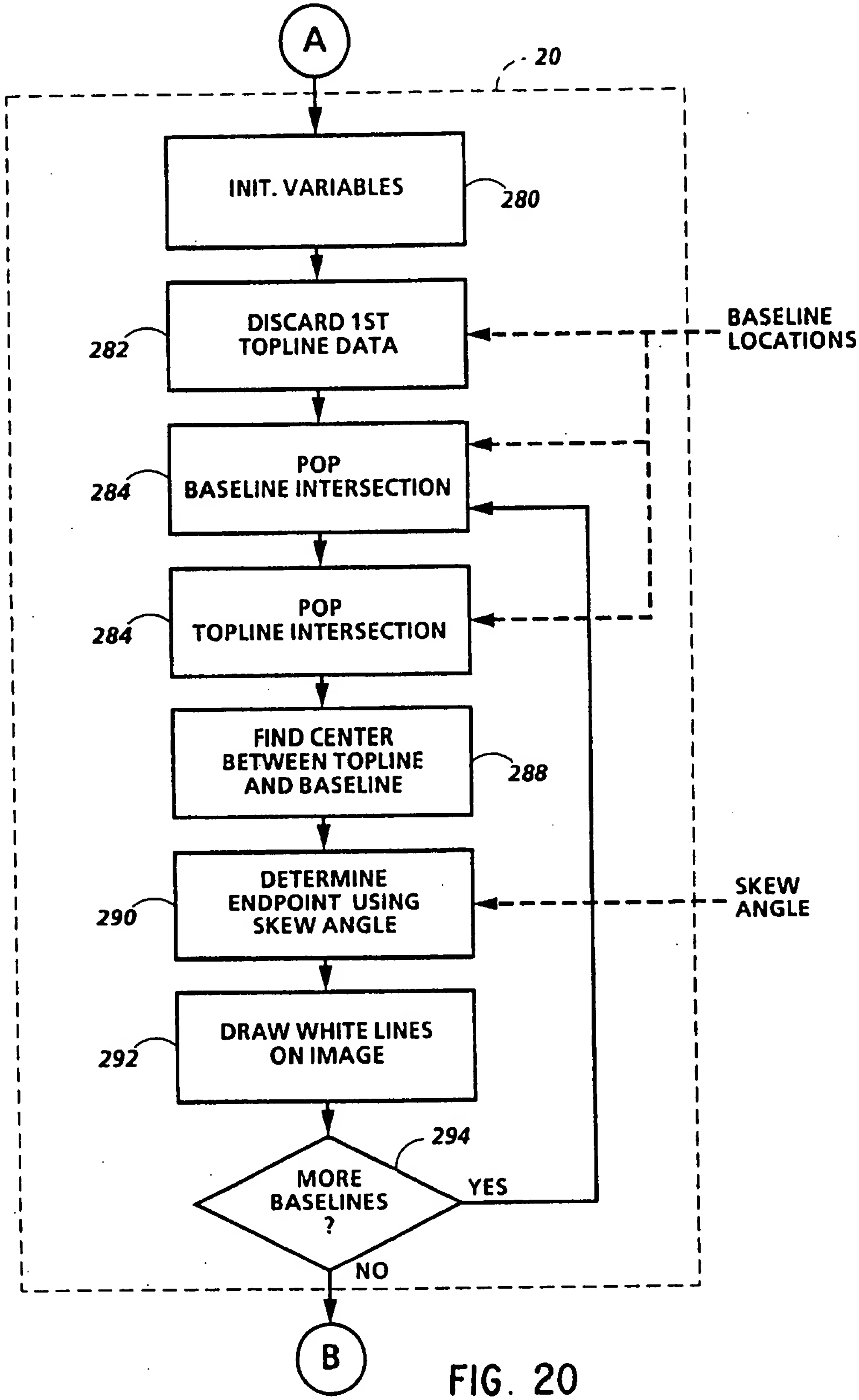

A
20
INIT. VARIABLES
280
DISCARD 1ST
TOPLINE DATA
282
BASELINE
LOCATIONS
POP
BASELINE INTERSECTION
284
POP
TOPLINE INTERSECTION
284
FIND CENTER
BETWEEN TOPLINE
AND BASELINE
288
DETERMINE
ENDPOINT USING
SKEW ANGLE
290
SKEW
ANGLE
DRAW WHITE LINES
ON IMAGE
292
MORE
BASELINES
?
294
YES
NO
B

FIG. 20

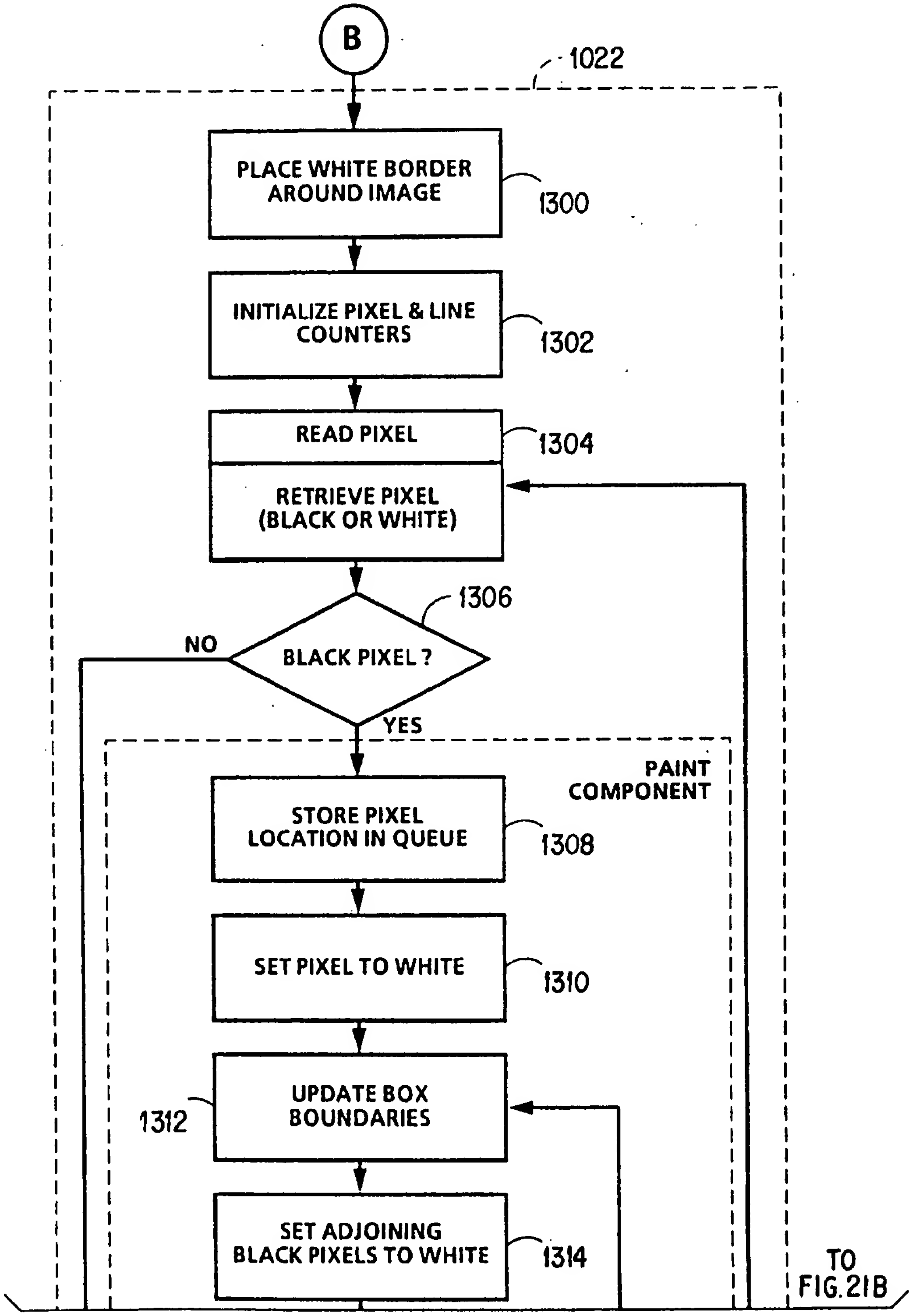

B
1022
PLACE WHITE BORDER AROUND IMAGE
1300
INITIALIZE PIXEL & LINE COUNTERS
1302
READ PIXEL
1304
RETRIEVE PIXEL (BLACK OR WHITE)
1306
NO
BLACK PIXEL ?
YES
PAINT COMPONENT
STORE PIXEL LOCATION IN QUEUE
1308
SET PIXEL TO WHITE
1310
UPDATE BOX BOUNDARIES
1312
SET ADJOINING BLACK PIXELS TO WHITE
1314
TO FIG.21B

FIG. 21A

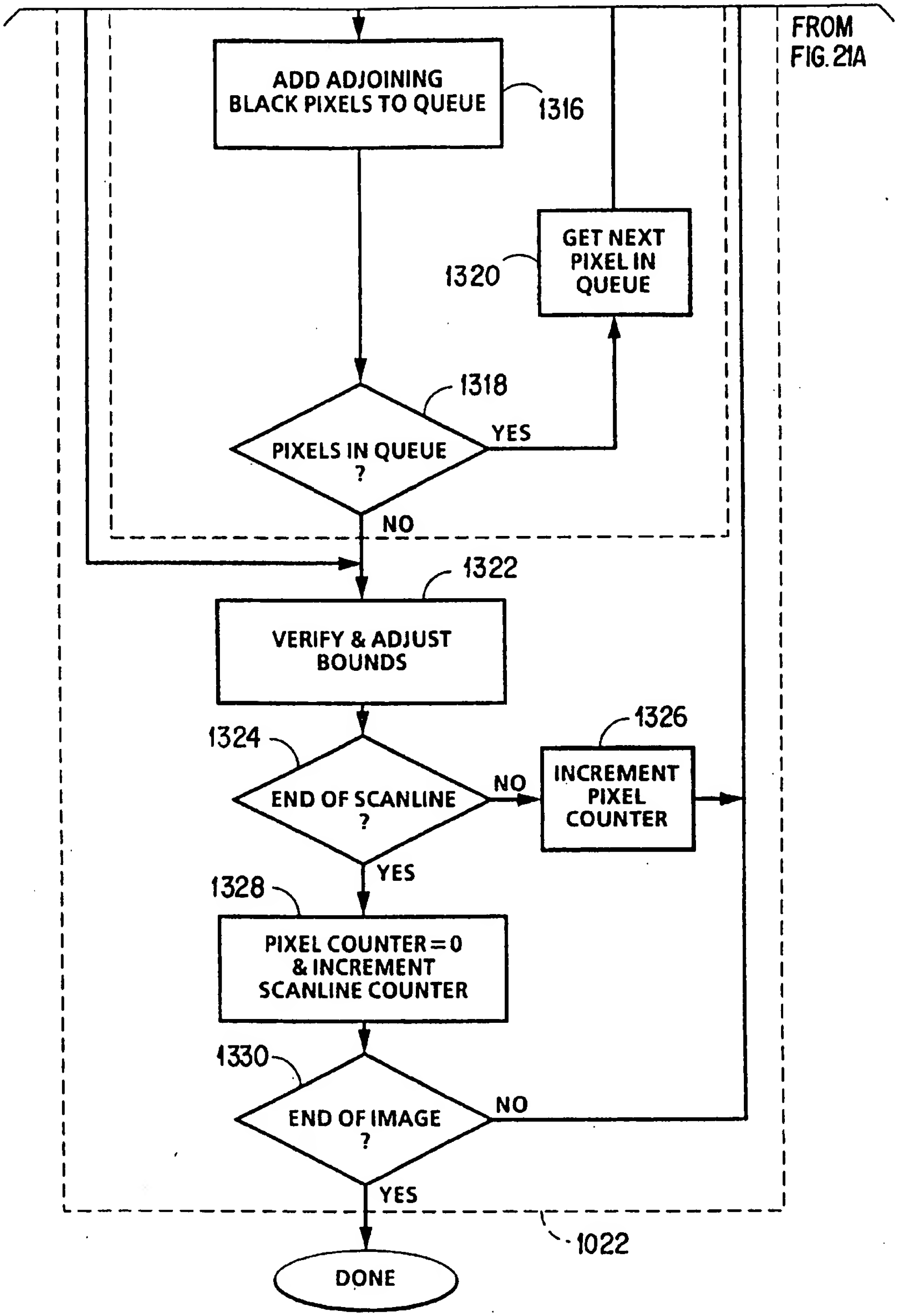
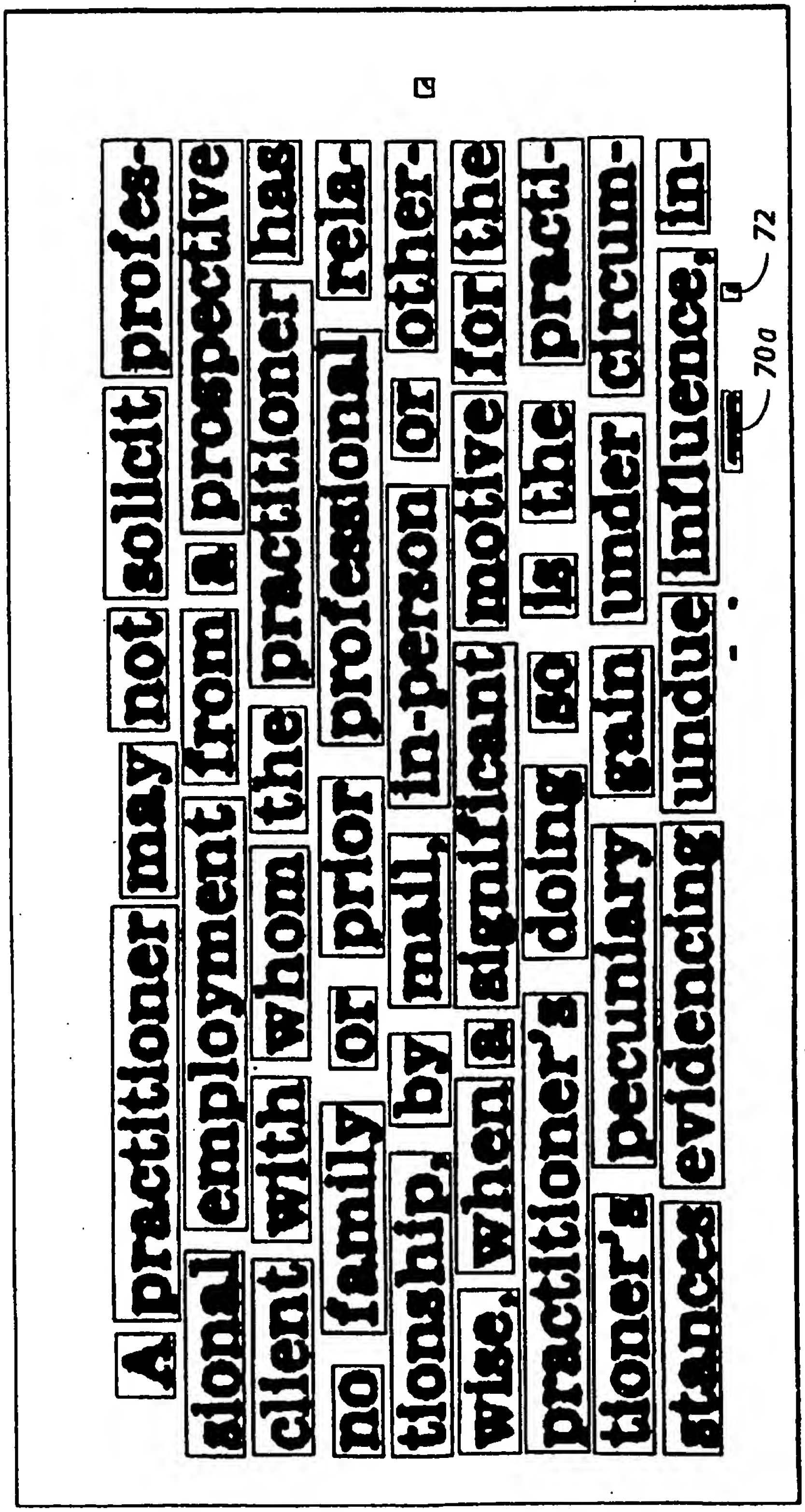

FROM FIG. 21A
ADD ADJOINING BLACK PIXELS TO QUEUE
1316
GET NEXT PIXEL IN QUEUE
1320
1318
PIXELS IN QUEUE ?
YES
NO
1322
VERIFY & ADJUST BOUNDS
1326
1324
END OF SCANLINE ?
NO
INCREMENT PIXEL COUNTER
YES
1328
PIXEL COUNTER=0 & INCREMENT SCANLINE COUNTER
1330
END OF IMAGE ?
NO
YES
1022
DONE

FIG. 21B



A practitioner may not solicit profes-
sional employment from a prospective
client with whom the practitioner has
no family or prior professional rela-
tionship, by mail, in-person or other-
wise, when a significant motive for the
practitioner's doing so is the practi-
tioner's pecuniary gain under circum-
stances evidencing undue influence, in-
72
70a

FIG. 22

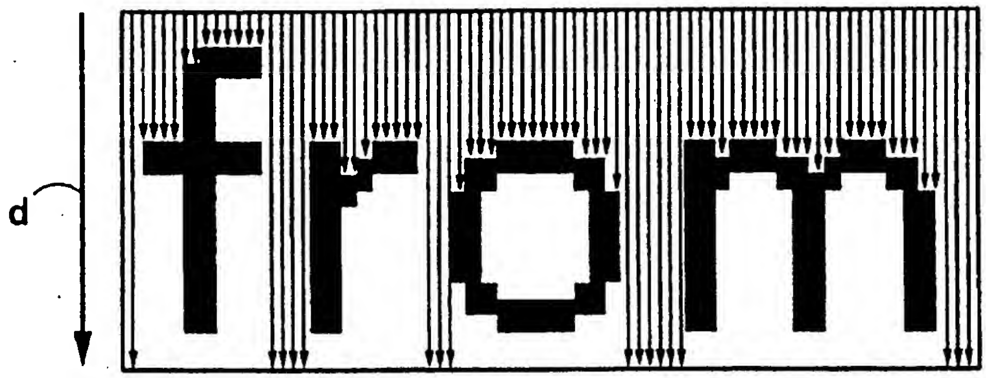


FIG. 23A

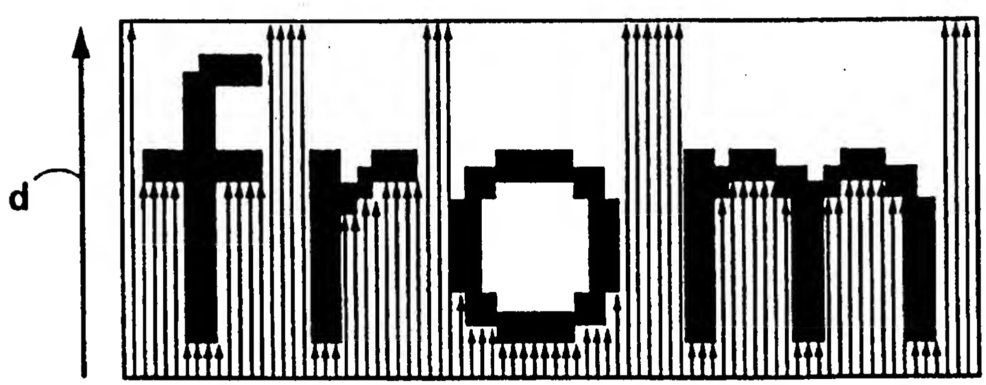


FIG. 23B

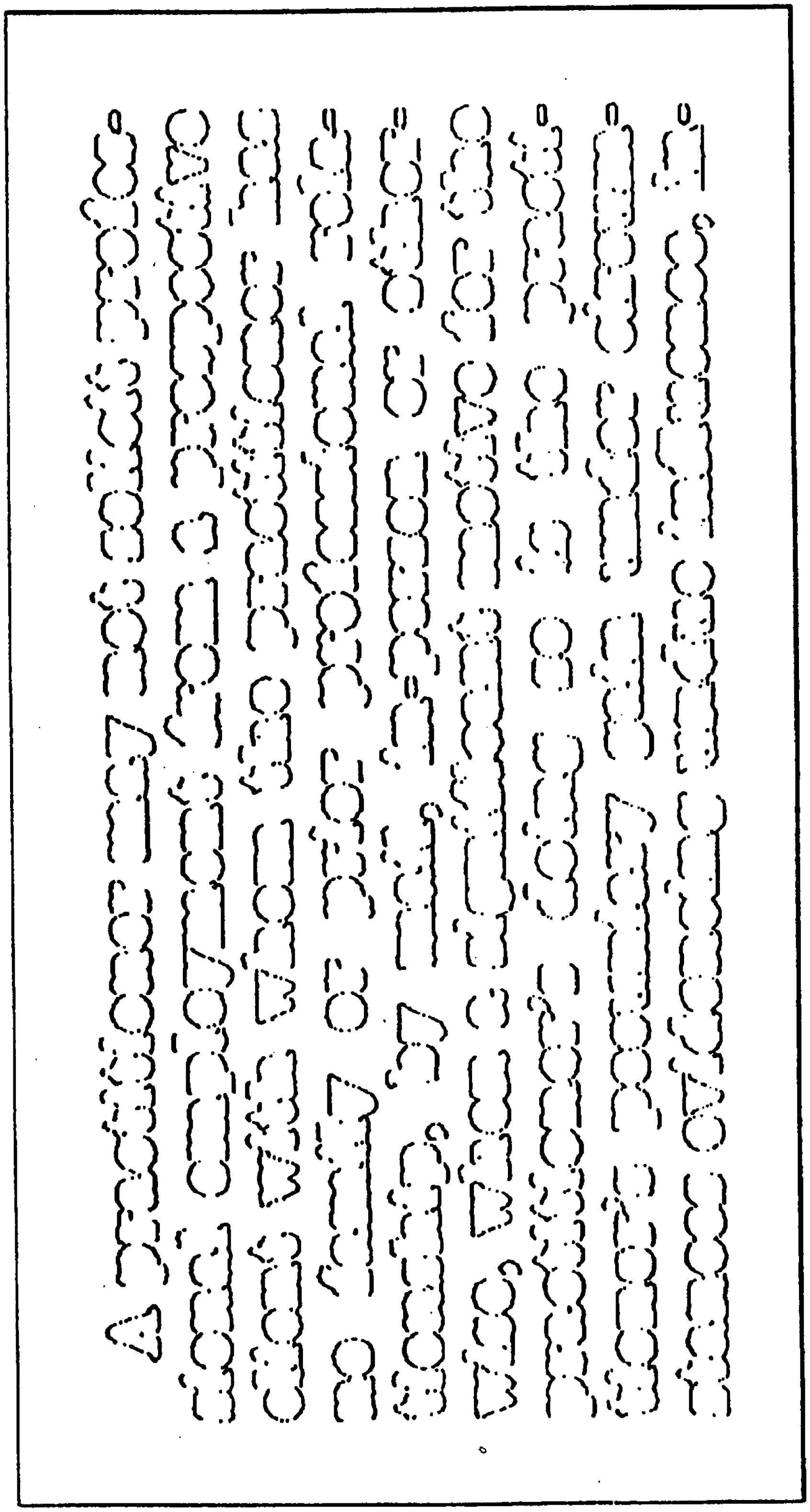


FIG. 24

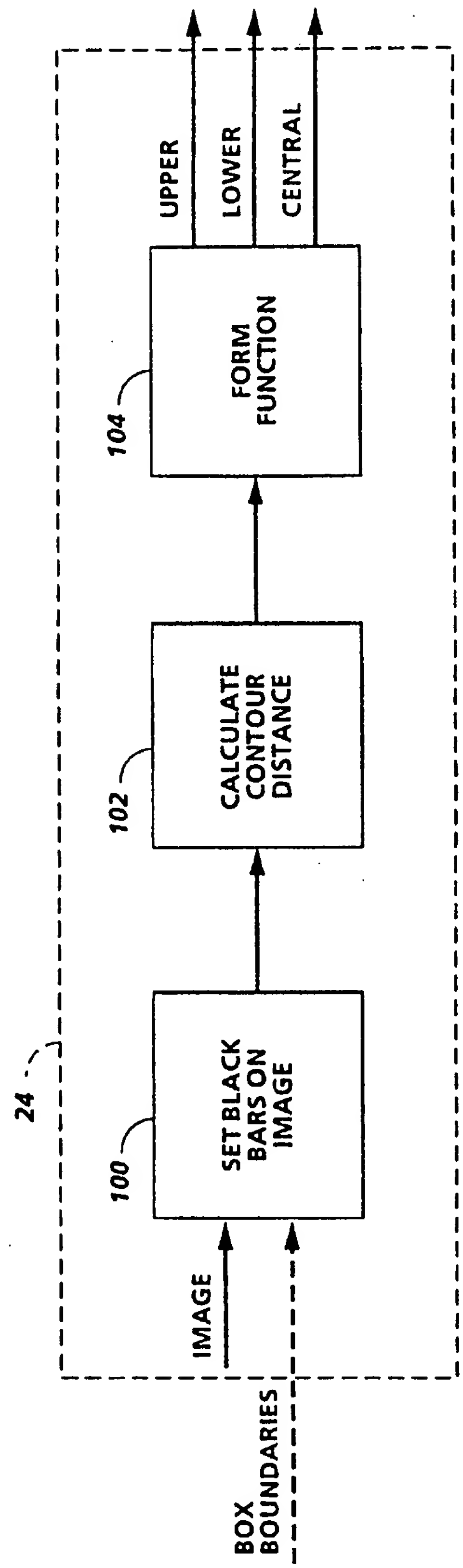

24
IMAGE
BOX BOUNDARIES
100
SET BLACK BARS ON IMAGE
102
CALCULATE CONTOUR DISTANCE
104
FORM FUNCTION
UPPER
LOWER
CENTRAL

FIG. 25

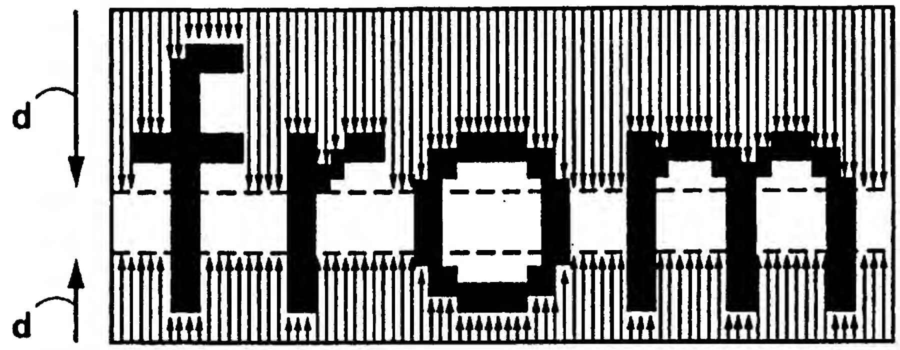


FIG. 26A

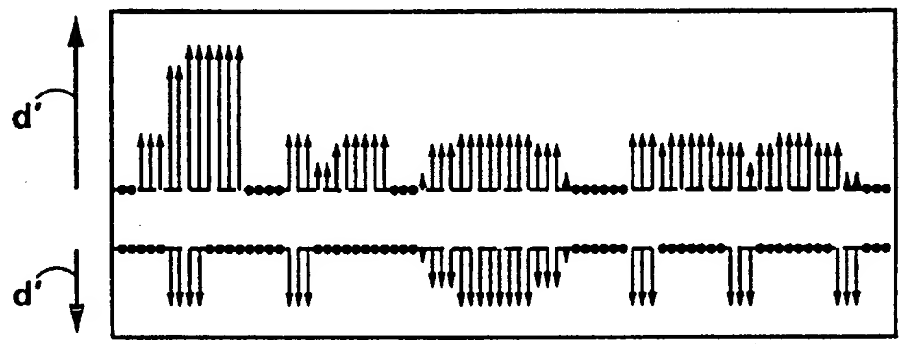
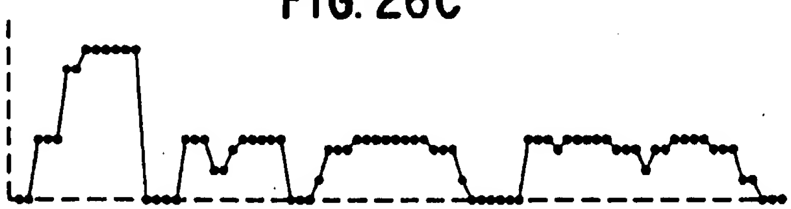


FIG. 26B

FIG. 26C



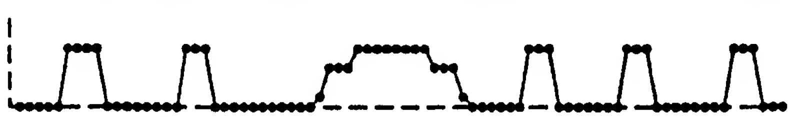


FIG. 26D

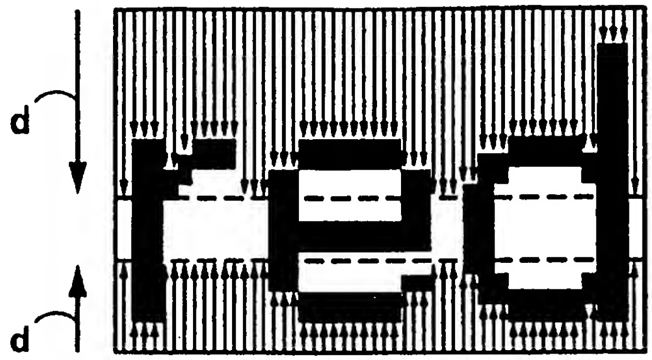


FIG. 27A

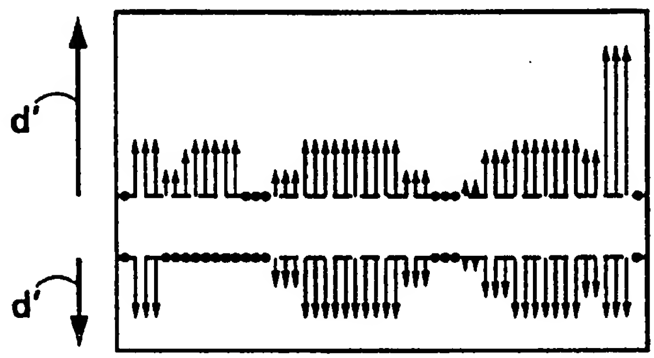
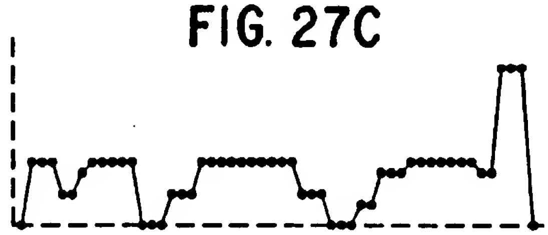


FIG. 27B

FIG. 27C



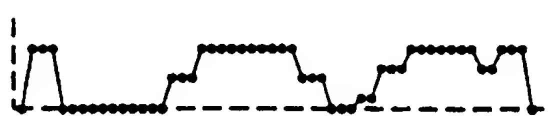


FIG. 27D

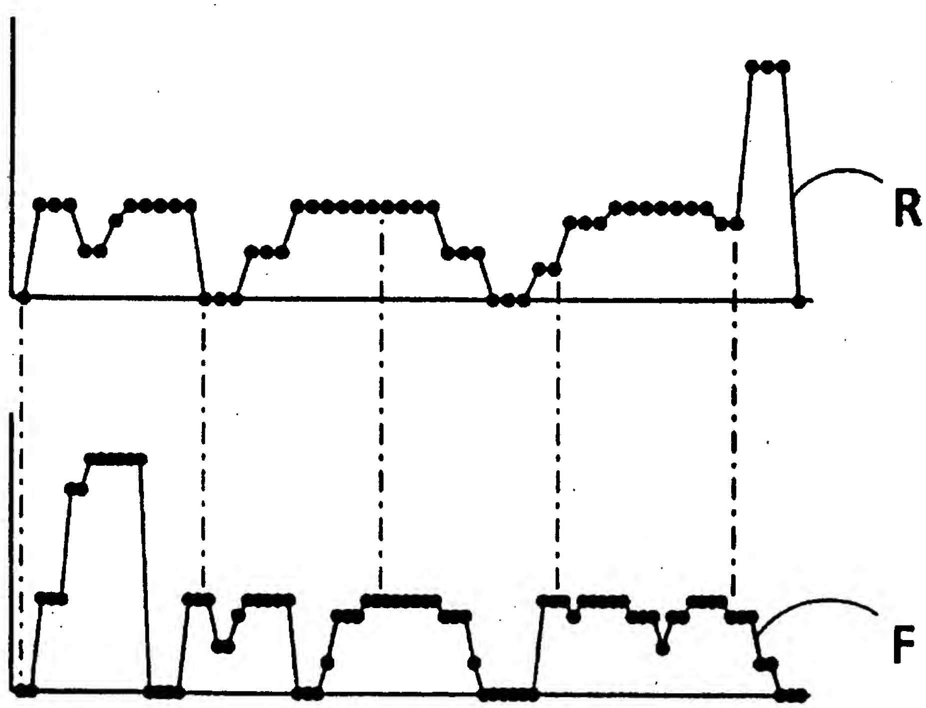


FIG. 28

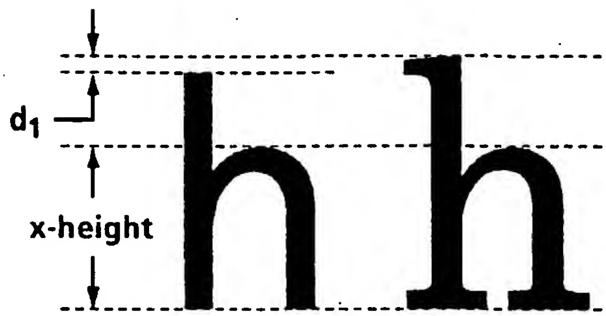


FIG. 29A

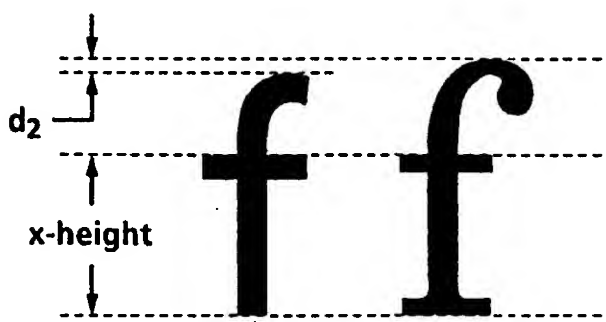


FIG. 29B

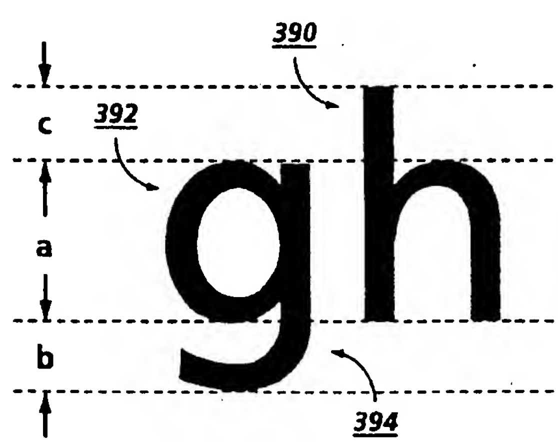


FIG. 29C

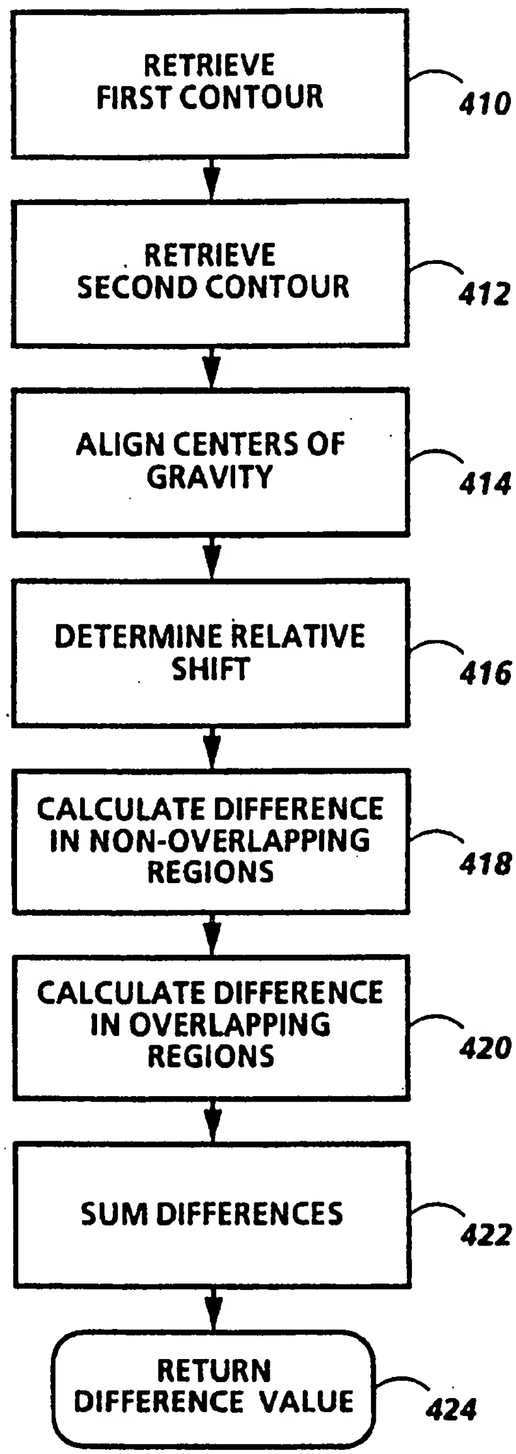

RETRIEVE FIRST CONTOUR
410
RETRIEVE SECOND CONTOUR
412
ALIGN CENTERS OF GRAVITY
414
DETERMINE RELATIVE SHIFT
416
CALCULATE DIFFERENCE IN NON-OVERLAPPING REGIONS
418
CALCULATE DIFFERENCE IN OVERLAPPING REGIONS
420
SUM DIFFERENCES
422
RETURN DIFFERENCE VALUE
424

FIG. 30

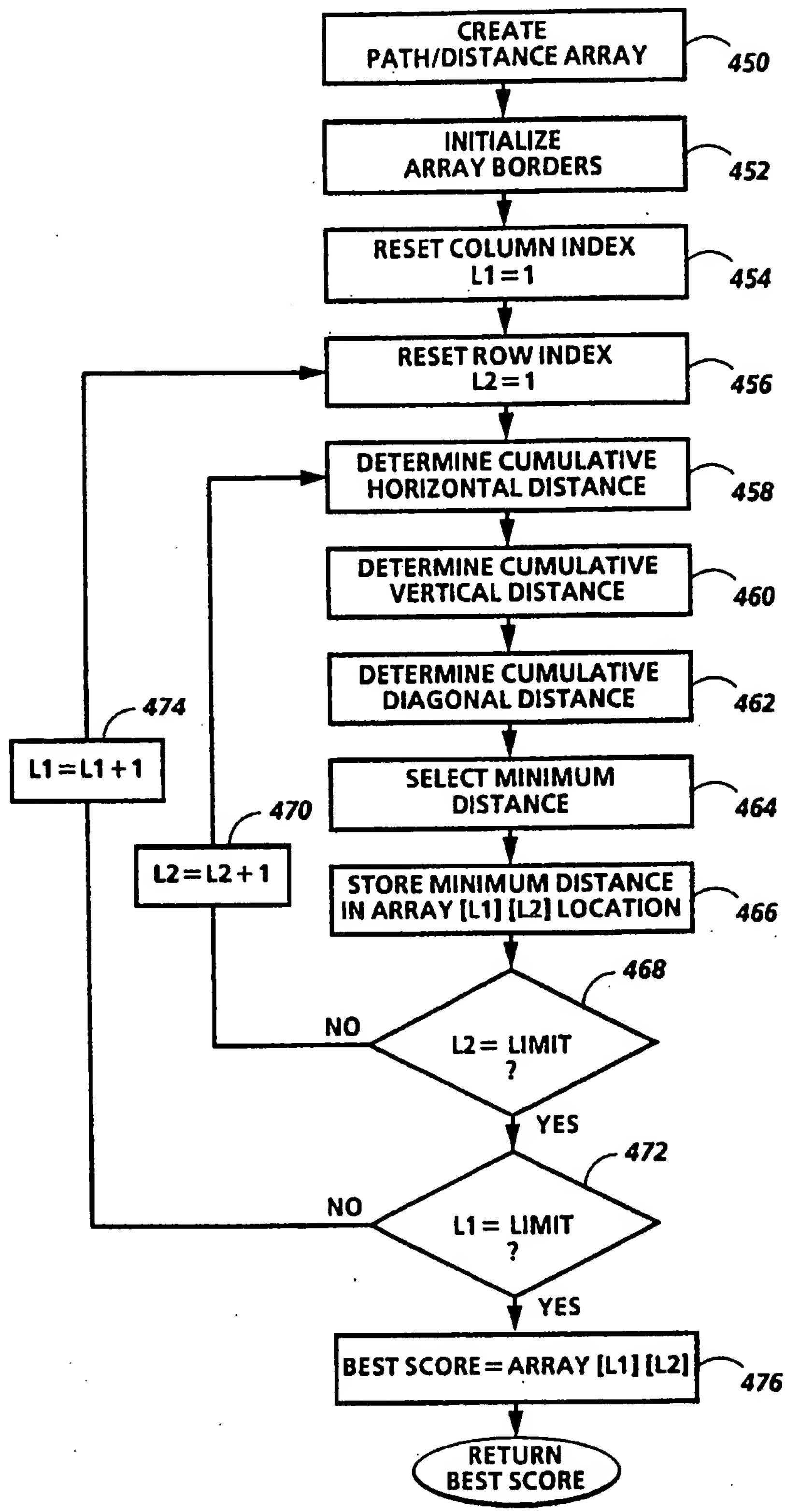

CREATE PATH/DISTANCE ARRAY
450
INITIALIZE ARRAY BORDERS
452
RESET COLUMN INDEX L1=1
454
RESET ROW INDEX L2=1
456
DETERMINE CUMULATIVE HORIZONTAL DISTANCE
458
DETERMINE CUMULATIVE VERTICAL DISTANCE
460
DETERMINE CUMULATIVE DIAGONAL DISTANCE
462
SELECT MINIMUM DISTANCE
464
STORE MINIMUM DISTANCE IN ARRAY [L1] [L2] LOCATION
466
L2= LIMIT ?
468
NO
YES
L2=L2+1
470
L1= LIMIT ?
472
NO
YES
L1=L1+1
474
BEST SCORE=ARRAY [L1] [L2]
476
RETURN BEST SCORE

FIG. 31

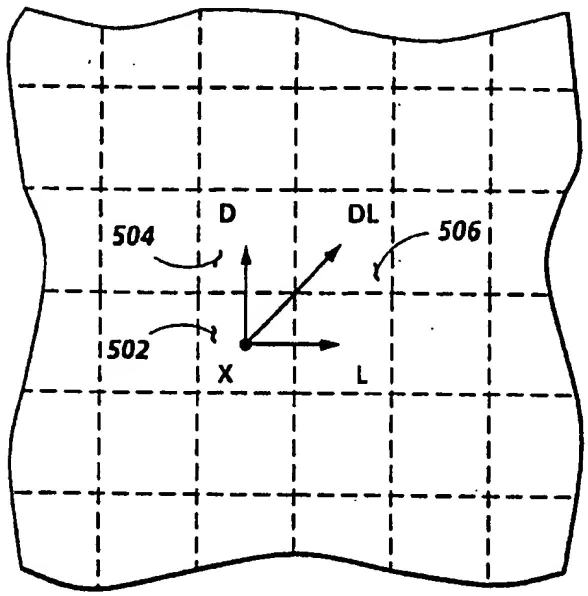


FIG. 32A

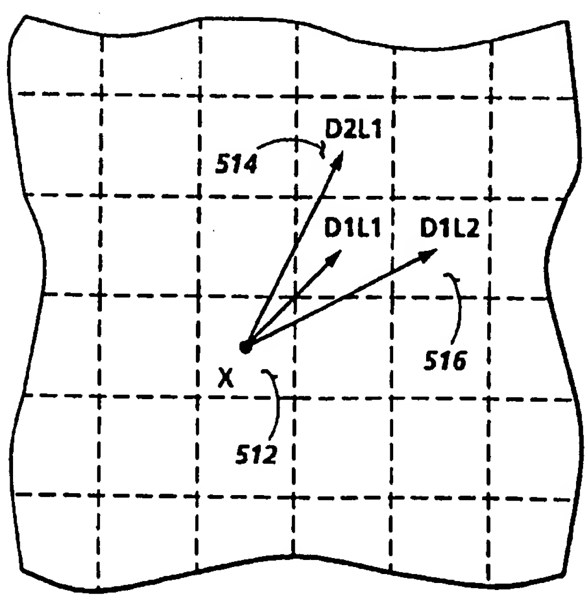


FIG. 32B

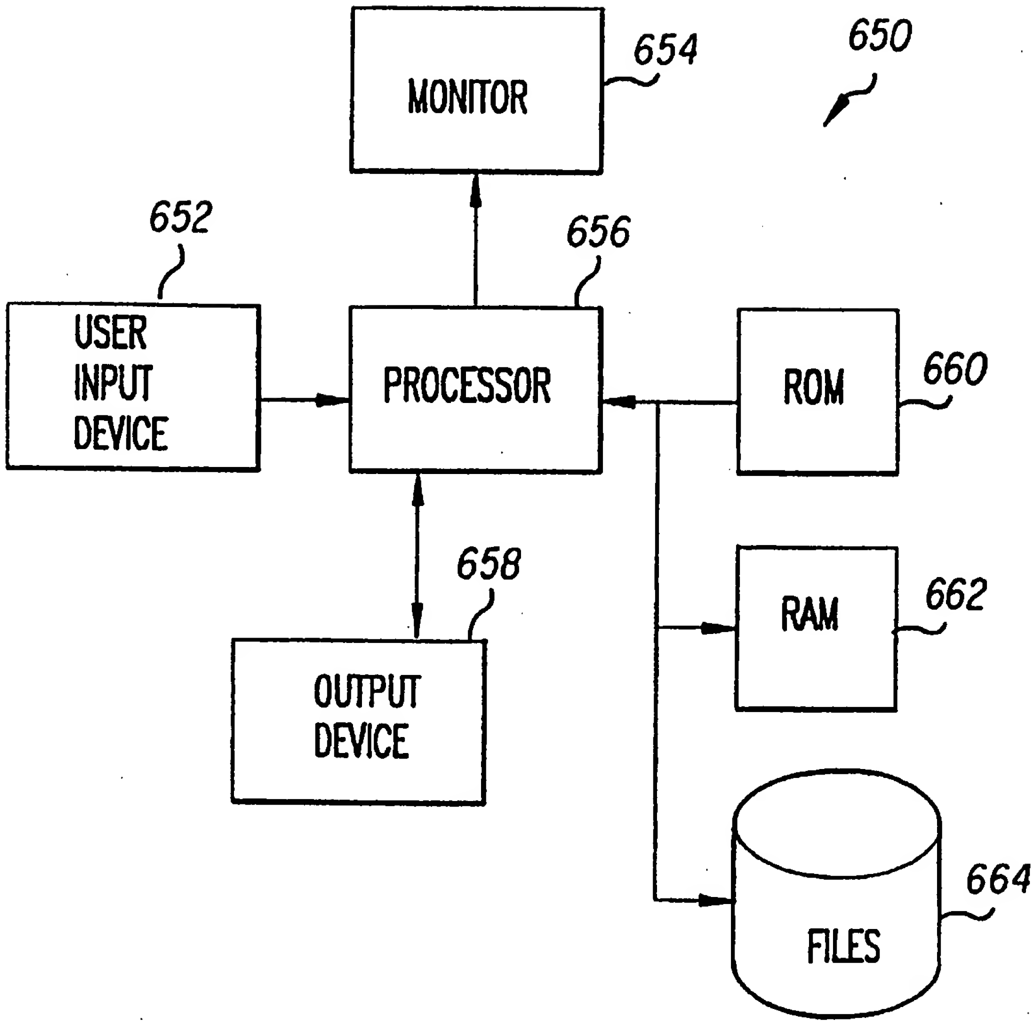

MONITOR
654
650
652
656
USER INPUT DEVICE
PROCESSOR
ROM
660
658
RAM
662
OUTPUT DEVICE
664
FILES

FIG. 33

METHOD AND APPARATUS FOR SUMMARIZING A DOCUMENT WITHOUT DOCUMENT IMAGE DECODING

This is a continuation of application Ser. No. 07/794,543 filed Nov. 19, 1991, now abandoned.

BACKGROUND OF THE INVENTION

1. Cross-References to Related Applications

The following concurrently filed and related U.S. applications are hereby cross referenced and incorporated by reference in their entirety.

"Method for Determining Boundaries of Words in Text" to Huttenlocher et al., U.S. patent application Ser. No. 07/794,392.

"Detecting Function Words Without Converting a Document to Character Codes" to Bloomberg et al., U.S. patent application Ser. No. 07/794,190.

"A Method of Deriving Wordshapes for Subsequent Comparison" to Huttenlocher et al., U.S. patent application Ser. No. 07/794,391.

"Method and Apparatus for Determining the Frequency of Words in a Document Without Document Image Decoding" to Cass et al., U.S. patent application Ser. No. 07/795,173.

"Optical Word Recognition by Examination of Word Shape" to Huttenlocher et al., U.S. patent application Ser. No. 07/796,119, Published European Application No. 0543592, published May 26, 1993.

"A Method and Apparatus for Automatic Modification of Selected Semantically Significant Image Segments Within a Document Without Document Image Decoding" to Huttenlocher et al., U.S. patent application Ser. No. 07/795,174.

"Method for Comparing Word Shapes" to Huttenlocher et al., U.S. patent application Ser. No. 07/795,169.

"Method and Apparatus for Determining the Frequency of Phrase in a Document Without Document Image Decoding" to Withgott et al., U.S. patent application Ser. No. 07/794, 555 now U.S. Pat. No. 5,369,714.

2. Field of the Invention

This invention relates to improvements in methods and apparatuses for automatic document processing, and more particularly to improvements in methods and apparatuses for recognizing semantically significant words, characters, images, or image segments in a document image without first decoding the document image and automatically creating a summary version of the document contents.

3. Background

It has long been the goal in computer based electronic document processing to be able, easily and reliably, to identify, access and extract information contained in electronically encoded data representing documents; and to summarize and characterize the information contained in a document or corpus of documents which has been electronically stored. For example, to facilitate review and evaluation of the information content of a document or corpus of documents to determine the relevance of same for a particular user's needs, it is desirable to be able to identify the semantically most significant portions of a document, in terms of the information they contain; and to be able to present those portions in a manner which facilitates the user's recognition and appreciation of the document contents. However, the problem of identifying the significant portions within a document is particularly difficult when dealing with images of the documents (bitmap image data), rather than with code representations thereof (e.g., coded representations of text such as ASCII). As opposed to ASCII text files, which permit users to perform operations such as Boolean algebraic key word searches in order to locate text of interest, electronic documents which have been produced by scanning an original without decoding to produce document images are difficult to evaluate without exhaustive viewing of each document image, or without hand-crafting a summary of the document for search purposes. Of course, document viewing or creation of a document summary require extensive human effort.

On the other hand, current image recognition methods, particularly involving textual material, generally involve dividing an image segment to be analyzed into individual characters which are then deciphered or decoded and matched to characters in a character library. One general class of such methods includes optical character recognition (OCR) techniques. Typically, OCR techniques enable a word to be recognized only after each of the individual characters of the word have been decoded, and a corresponding word image retrieved from a library.

Moreover, optical character recognition decoding operations generally require extensive computational effort, generally have a non-trivial degree of recognition error, and often require significant amounts of time for image processing, especially with regard to word recognition. Each bitmap of a character must be distinguished from its neighbors, its appearance analyzed, and identified in a decision making process as a distinct character in a predetermined set of characters. Further, the image quality of the original document and noise inherent in the generation of a scanned image contribute to uncertainty regarding the actual appearance of the bitmap for a character. Most character identifying processes assume that a character is an independent set of connected pixels. When this assumption fails due to the quality of the image, identification also fails.

4. References

European patent application number 0-361-464 by Doi describes a method and apparatus for producing an abstract of a document with correct meaning precisely indicative of the content of the document. The method includes listing hint words which are preselected words indicative of the presence of significant phrases that can reflect content of the document, searching all the hint words in the document, extracting sentences of the document in which any one of the listed hint words is found by the search, and producing an abstract of the document by juxtaposing the extracted sentences. Where the number of hint words produces a lengthy excerpt, a morphological language analysis of the abstracted sentences is performed to delete unnecessary phrases and focus on the phrases using the hint words as the right part of speech according to a dictionary containing the hint words.

"A Business Intelligence System" by Luhn, IBM Journal, October 1958 describes a system which in part, autoabstracts a document, by ascertaining the most frequently occurring words (significant words) and analyzes all sentences in the text containing such words. A relative value of the sentence significance is then established by a formula

which reflects the number of significant words contained in a sentence and the proximity of these words to each other within the sentence. Several sentences which rank highest in value of significance are then extracted from the text to constitute the auto-abstract.

SUMMARY OF THE INVENTION

Accordingly, it is an object of the invention to provide a method and apparatus for automatically excerpting and summarizing a document image without decoding or otherwise understanding the contents thereof.

It is another object of the invention to provide a method and apparatus for automatically generating ancillary document images reflective of the contents of an entire primary document image.

It is another object of the invention to provide a method and apparatus of the type described for automatically extracting summaries of material and providing links from the summary back to the original document.

It is another object of the invention to provide a method and apparatus of the type described for producing Braille document summaries or speech synthesized summaries of a document.

It is another object of the invention to provide a method and apparatus of the type described which is useful for enabling document browsing through the development of image gists, or for document categorization through the use of lexical gists.

It is another object of the invention to provide a method and apparatus of the type described that does not depend upon statistical properties of large, pre-analyzed document corpora.

The invention provides a method and apparatus for segmenting an undecoded document image into undecoded image units, identifying semantically significant image units based on an evaluation of predetermined image characteristics of the image units, without decoding the document image or reference to decoded image data, and utilizing the identified significant image units to create an ancillary document image of abbreviated information content which is reflective of the subject matter content of the original document image. In accordance with one aspect of the invention, the ancillary document image is a condensation or summarization of the original document image which facilitates browsing. In accordance with another aspect of the invention, the identified significant image units are presented as an index of key words, which may be in decoded form, to permit document categorization.

Thus, in accordance with one aspect of the invention, a method is presented for excerpting information from a document image containing word image units. According to the invention, the document image is segmented into word image units (word units), and the word units are evaluated in accordance with morphological image properties of the word units, such as word shape. Significant word units are then identified, in accordance with one or more predetermined or user selected significance criteria, and the identified significant word units are outputted.

In accordance with another aspect of the invention, an apparatus is provided for excerpting information from a document containing a word unit text. The apparatus includes an input means for inputting the document and producing a document image electronic representation of the document, and a data processing system for performing data driven processing and which comprises execution processing means for performing functions by executing program instructions in a predetermined manner contained in a memory means. The program instructions operate the execution processing means to identify significant word units in accordance with a predetermined significance criteria from morphological properties of the word units, and to output selected ones of the identified significant word units. The output of the selected significant word units can be to an electrostatographic reproduction machine, a speech synthesizer means, a Braille printer, a bitmap display, or other appropriate output means.

These and other objects, features and advantages of the invention will be apparent to those skilled in the art from the following detailed description of the invention, when read in conjunction with the accompanying drawings and appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

A preferred embodiment of the invention is illustrated in the accompanying drawing, in which:

FIG. 1 is a flow chart of a method of the invention;

FIG. 2 is a block diagram of an apparatus according to the invention for carrying out the method of FIG. 1;

FIG. 3 is a flow chart of a preferred embodiment of a method according to the invention for detecting function words in a scanned document image without first converting the document image to character codes;

FIGS. 4A–4F show three sets of character ascender structuring elements where: FIGS. 4A–4B show a set of character ascender structuring elements of height 3 and length 5, where the solid dots are ON pixels along the bottom row and along one side column and there are one or more OFF pixels in a remaining location preferably separated from the ON pixels; FIGS. 4C–4D show a set of character ascender structuring elements of height 4 and length 5; and FIGS. 4E–4F show a set of character ascender structuring elements of height 5 and length 5.

FIGS. 5A–5F show three sets of character descender structuring elements where: FIGS. 5A–5B show a set of character descender structuring elements of height 3 and length 5; FIGS. 5C–5D show a set of character descender structuring elements of height 4 and length 5; and FIGS. 5E–5F show a set of character descender structuring elements of height 5 and length 5;

FIG. 6 shows a horizontal structuring element of length 5;

FIG. 7 shows a block system diagram of the arrangement of system components forming a word shape recognition system;

FIG. 8 shows a block system diagram for identifying equivalence classes of image units; and

FIG. 9 shows a block system diagram for identifying significant image units.

FIG. 10 shows an image sample of example text over which the inventive process will be demonstrated;

FIG. 11 is a copy of a scanned image of the example text;

FIGS. 12A, 12B and 12C graphically illustrate the process used to determine the angle at which the example text is oriented in the image sample prior for further processing, while FIG. 12D shows graphs of the responses taken from the example text, which are used to determine the angle at which the example text is oriented in the image sample prior to further processing;

FIGS. 13A and 13B respectively show the derivation and use of a graph examining the sample image of the example text to determine baselines of text within the image;

FIGS. 14A and 14B are flowcharts illustrating the procedures executed to determine the baselines shown in FIG. 13A;

FIG. 15 shows the scanned image of the example text with baselines indicated thereon after derivation from the data shown in FIGS. 13A and 13B;

FIG. 16 is a flowchart illustrating the steps used in the application of a median filter to the image of FIG. 10;

FIG. 17 is an enlarged pictorial representation of a portion of the image of FIG. 10, illustrating the application of the median filter;

FIG. 18 demonstrates the resulting image after application of a median filter, a process known herein as blobifying, to the scanned image of the example text, which tends to render character strings as a single set of connected pixels;

FIG. 19 shows a subsequent step in the process, in which lines of white pixels are added to the blurred image to clearly delineate a line of character strings from adjacent lines of character strings;

FIG. 20 is a flowchart illustrating the steps required to add the white lines of FIG. 19;

FIGS. 21A and 21B are flowcharts representing the procedure which is followed to segment the image data in accordance with the blurred image of FIG. 18;

FIG. 22 shows the sample text with bounding boxes placed around each word group in a manner which uniquely identifies a subset of image pixels containing each character string;

FIGS. 23A and 23B illustrate derivation of a single independent value signal, using the example word "from", which appears in the sample image of example text;

FIG. 24 illustrates the resulting contours formed by the derivation process illustrated in FIGS. 23A and 23B;

FIG. 25 illustrates the steps associated with deriving the word shape signals;

FIGS. 26A, 26B, 26C and 26D illustrate derivation of a single independent value signal, using the example word "from";

FIGS. 27A, 27B, 27C and 27D illustrate derivation of a single independent value signal, using the example word "red", which does not appear in the sample image of example text;

FIG. 28 shows a simple comparison of the signals derived for the words "red" and "from" using a signal normalization method;

FIGS. 29A, 29B, and 29C illustrate the details of the discrepancy in font height, and the method for normalization of such discrepancies;

FIG. 30 is a flowchart detailing the steps used for one method of determining the relative difference between word shape contours;

FIG. 31 is a flowchart detailing the steps of a second method for determining the relative difference between word shape contours;

FIGS. 32A and 32B are respective illustrations of the relationship between the relative difference values calculated and stored in an array, for both a non-slope-constrained and a slope-constrained comparison; and

FIG. 33 is a block diagram of a preferred embodiment of an apparatus according to the invention for detecting function words in a scanned document image without first converting the document image to character codes;

The Appendix contains source code listings for a series of image manipulation and signal processing routines which have been implemented to demonstrate the functionality of the present invention. Included in the Appendix are four sections which are organized as follows:

Section A, beginning at page 1, comprises the declarative or "include" files which are commonly shared among the functional code modules;

Section B, beginning at page 26, includes the listings for a series of library type functions used for management of the images, error reporting, argument parsing, etc.;

Section C, beginning at page 42, comprises numerous variations of the word shape comparison code, and further includes code illustrating alternative comparison techniques than those specifically cited in the following description;

Section D, beginning at page 145, comprises various functions for the word shape extraction operations that are further described in the following description.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In contrast to prior techniques, such as those described above, the invention is based upon the recognition that scanned image files and character code files exhibit important differences for image processing, especially in data retrieval. The method of a preferred embodiment of the invention capitalizes on the visual properties of text contained in paper documents, such as the presence or frequency of linguistic terms (such as words of importance like "important", "significant", "crucial", or the like) used by the author of the text to draw attention to a particular phrase or a region of the text; the structural placement within the document image of section titles and page headers, and the placement of graphics; and so on. A preferred embodiment of the method of the invention is illustrated in the flow chart of FIG. 1, and an apparatus for performing the method is shown in FIG. 2. For the sake of clarity, the invention will be described with reference to the processing of a single document. However, it will be appreciated that the invention is applicable to the processing of a corpus of documents containing a plurality of documents. More particularly, the invention provides a method and apparatus for automatically excerpting semantically significant information from the data or text of a document based on certain morphological (structural) image characteristics of image units corresponding to units of understanding contained within the document image. The excerpted information can be used, among other things, to automatically create a document index or summary. The selection of image units for summarization can be based on frequency of occurrence, or predetermined or user selected selection criteria, depending upon the particular application in which the method and apparatus of the invention is employed.

The invention is not limited to systems utilizing document scanning. Rather, other systems such as a bitmap workstation (i.e., a workstation with a bitmap display) or a system using both bitmapping and scanning would work equally well for the implementation of the methods and apparatus described herein.

With reference first to FIG. 2, the method is performed on an electronic image of an original document 5, which may include lines of text 7, titles, drawings, figures 8, or the like, contained in one or more sheets or pages of paper 10 or other

tangible form. The electronic document image to be processed is created in any conventional manner, for example, by a conventional scanning means such as those incorporated within a document copier or facsimile machine, a Braille reading machine, or by an electronic beam scanner or the like. Such scanning means are well known in the art, and thus are not described in detail herein. An output derived from the scanning is digitized to produce undecoded bit mapped image data representing the document image for each page of the document, which data is stored, for example, in a memory **15** of a special or general purpose digital computer data processing system **13**. The data processing system **13** can be a data driven processing system which comprises sequential execution processing means **16** for performing functions by executing program instructions in a predetermined sequence contained in a memory, such as the memory **15**. The output from the data processing system **13** is delivered to an output device **17**, such as, for example, a memory or other form of storage unit; an output display **17A** as shown, which may be, for instance, a CRT display; a printer device **17B** as shown, which may be incorporated in a document copier machine or a Braille or standard form printer; a facsimile machine, speech synthesizer or the like.

Through use of equipment such as illustrated in FIG. **2**, the identified word units are detected based on significant morphological image characteristics inherent in the image units, without first converting the scanned document image to character codes.

The method by which such image unit identification may be performed is described with reference now to FIG. **1**. The first phase of the image processing technique of the invention involves a low level document image analysis in which the document image for each page is segmented into undecoded information containing image units (step 20) using conventional image analysis techniques; or, in the case of text documents, preferably using the bounding box method described in copending U.S. patent application Ser. No. 07/794,392 filed concurrently herewith by Huttenlocher and Hopcroft, and entitled "Method for Determining Boundaries of Words in Text." The locations of and spatial relationships between the image units on a page are then determined (step 25). For example, an English language document image can be segmented into word image units based on the relative difference in spacing between characters within a word and the spacing between words. Sentence and paragraph boundaries can be similarly ascertained. Additional region segmentation image analysis can be performed to generate a physical document structure description that divides page images into labelled regions corresponding to auxiliary document elements like figures, tables, footnotes and the like. Figure regions can be distinguished from text regions based on the relative lack of image units arranged in a line within the region, for example. Using this segmentation, knowledge of how the documents being processed are arranged (e.g., left-to-right, top-to-bottom), and, optionally, other inputted information such as document style, a "reading order" sequence for word images can also be generated. The term "image unit" is thus used herein to denote an identifiable segment of an image such as a number, character, glyph, symbol, word, phrase or other unit that can be reliably extracted. Advantageously, for purposes of document review and evaluation, the document image is segmented into sets of signs, symbols or other elements, such as words, which together form a single unit of understanding. Such single units of understanding are generally characterized in an image as being separated by a spacing greater than that which separates the elements forming a unit, or by some predetermined graphical emphasis, such as, for example, a surrounding box image or other graphical separator, which distinguishes one or more image units from other image units in the scanned document image. Such image units representing single units of understanding will be referred to hereinafter as "word units."

Advantageously, a discrimination step 30 is next performed to identify the image units which have insufficient information content to be useful in evaluating the subject matter content of the document being processed. One preferred method is to use the morphological function or stop word detection techniques disclosed in the copending U.S. patent application Ser. No. 07/794,190 filed concurrently herewith by D. Bloomberg et al., and entitled "Detecting Function Words Without Converting a Document to Character Codes".

The method of identification of image units which have insufficient information content by determining function words without converting the document to character codes is shown in FIG. **3**. The following definitions are used to describe this method:

A binary image contains pixels that are either ON or OFF. Binary images are manipulated according to a number of operations wherein one or more source images are mapped onto a destination image. The results of such operations are generally referred to as images.

A morphological operation refers to an operation on a pixelmap image (a source image), that uses a local rule at each pixel to create another pixelmap image, the destination image. This rule depends both on the type of the desired operation to perform as well as on the chosen structuring element.

A structuring element (SE) refers to an image object of typically (but not necessarily) small size and simple shape that probes the source image and extracts various types of information from it via the chosen morphological operation. FIGS. **4** and **5** show SEs where a solid circle is a hit, and an open circle is a miss. The center position is denoted by a cross. Squares that have neither solid nor open circles are "don't cares"; their value in the image (ON or OFF) is not probed. A binary SE is used to probe binary images in a binary morphological operation that operates on binary input images and creates an output binary image. The SE is defined by a center location and a number of pixel locations, each normally having a defined value (ON or OFF). The pixels defining the SE do not have to be adjacent each other. The center location need not be at the geometrical center of the pattern; indeed it need not even be inside the pattern. A solid SE refers to an SE having a periphery within which all pixels are ON. For example, a solid 2×2 SE is a 2×2 square of ON pixels. A solid SE need not be rectangular. A horizontal SE is generally one row of ON pixels and a vertical SE is generally one column of ON pixels of selected size. A hit-miss SE refers to an SE that specifies at least one ON pixel and at least one OFF pixel.

AND, OR and XOR are logical operations carried out between two images on a pixel-by-pixel basis.

NOT is a logical operation carried out on a single image on a pixel-by-pixel basis.

EXPANSION is scale operation characterized by a scale factor N, wherein each pixel in a source image becomes an N×N square of pixels, all having the same value as the original pixel.

REDUCTION is a scale operation characterized by a scale factor N in a threshold level M. REDUCTION with scale=N entails dividing the source image into N×N squares

of pixels, mapping each such square in the source image to a single pixel on the destination image. The value for the pixel in the destination image is determined by the threshold level M, which is a number between I and N^2. If the number of ON pixels in the pixel square is greater or equal to M, the destination pixel is ON, otherwise it is OFF.

EROSION is a morphological operation wherein a given pixel in the destination image is turned ON if and only if the result of superimposing the SE center on the corresponding pixel location in the source image results in a match between all ON and OFF pixels in the SE and the underlying pixels in the source image. An EROSION will give one pixel in the destination image for every match. That is, at each pixel, it outputs 1 if the SE (shifted and centered at that pixel) is totally contained inside the original image foreground, and outputs 0 otherwise. Note that EROSION usually refers to operations using a SE with only hits and more generally matching operations with both hits and misses (often called a hit-miss transform). The term EROSION is used herein to include matching operations with both hits and misses, thus the hit-miss transform is the particular type of EROSION used herein.

DILATION is a morphological operation wherein a given pixel in the source image being ON causes the SE to be written into the destination image with the SE center at the corresponding location in the destination image. The SEs used for DILATION typically have no OFF pixels. The DILATION draws the SE as a set of pixels in the destination image for each pixel in the source image. Thus, the output image is the union of all shifted versions of the SE translated at all 1-pixels of the original image.

FillClip is a morphological operation where one image is used as a seed and is grown morphologically, clipping it at each growth step to the second image. For example, a fillClip could include a DILATION followed by logically ANDing the DILATION result with another image.

OPENING is a morphological operation that uses an image and a structuring element and consists of an EROSION followed by a DILATION. The result is to replicate the structuring element in the destination image for each match in the source image.

CLOSING is a morphological operation using an image and a structuring element. It includes a DILATION followed by an EROSION of the image by a structuring element. A CLOSE of an image is equivalent to the bit inverse of an OPEN on the (bit inverse) background.

UNION is a bitwise OR between two images. An intersection is a bitwise AND between two images.

Blurring is a DILATION of an image by a structuring element(s) consisting of two or more hits.

A mask refers to an image, normally derived from an original or source image, that contains substantially solid regions of ON pixels corresponding to regions of interest in the original image. The mask may also contain regions of ON pixels that do not correspond to regions of interest.

The various operations defined above are sometimes referred to in noun, adjective, and verb forms. For example, references to DILATION (noun form) may be in terms of DILATING the image or the image being DILATED (verb forms) or the image being subjected to a DILATION operation (adjective form). No difference in meaning is intended.

Morphological operations have several specific properties that simplify their use in the design of appropriate procedures. First, they are translationally invariant. A sideway shift of the image before transforming does not change the result, except to shift the result as well. Operations that are translationally invariant can be implemented with a high degree of parallelism, in that each point in the image is treated using the same rule. In addition, morphological operations satisfy two properties that make it easy to visualize their geometrical behavior. First, EROSION, DILATION, OPEN and CLOSE are increasing, which means that if image 1 is contained in image 2, then any of these morphological operations on image 1 will also be contained in the morphological operation on image 2. Second, a CLOSE is extensive and OPEN is antiextensive. This means that the original image is contained in the image transformed by CLOSE and the image transformed by OPEN is contained in the original image. The DILATION and EROSION operations are also extensive and anti-extensive, respectively, if the center of the structuring element is located on a hit.

The OPEN and CLOSE operations also satisfy two more morphological properties:

(1) The result of the operation is independent of the position of the center of the structuring element.

(2) The operation is idempotent, which means that reapplying the OPEN or CLOSE to the resulting image will not change it.

An image unit means an identifiable segment of an image such as a word, number, character, glyph or other units that can be extracted reliably and have an underlying linguistic structure.

The term significant and its derivatives are used in this description to indicate the importance of particular characteristics of an image unit. An image unit with significant characteristics becomes a significant image unit in that it contains high value information which can be used for further processing of the document image. Significant characteristics of image units include a variety of classifiers such as length, width, location on a page of the document image, font, typeface and measurement by other parameters including, but not limited to: one or more cross-sections of a box (a cross-section being a sequence of ON or OFF pixels); a number of ascenders associated with an image unit; a number of descenders associated with an image unit; average pixel density in an image unit; a length of a topline contour of an image unit, including peaks and troughs; a length of a base contouring of the image units, including peaks and troughs; and the location of image units with respect to neighboring image units, e.g., vertical position and horizontal inter-image unit spacing.

Referring to FIG. 3, the method for detecting function words in a scanned document image without first converting the document image to character codes will be described. An image of a page of a document is scanned in step 302 and the image is segmented into image units in step 304 by using either a conventional image analysis techniques or by using first a technique to determine baselines of image units and then second a technique for providing bounding boxes around image units (see U.S. patent application Ser. No. 07/794,391 entitled "A Method of Deriving Wordshapes for Subsequent Comparison" by Huttenlocher et al.)

In step 306, a length and height of each image unit in the image is determined. Short image units are determined in step 308 as image units of no more than a predetermined number of characters, preferably three characters or less in length. In step 310, image units which are not short image units are deleted from the image. In step 312, the image is blurred or smeared in a horizontal direction although the image units are not smeared together. This can be accomplished for example by CLOSING the image with a hori-

zontal structuring element such as the structuring element of length 5 (i.e., 5 pixels) shown in FIG. 6. The length of the horizontal structuring element used to blur the x-height characters in the image is dependent upon the width of the character type being used. Furthermore, other configurations of structuring elements may be used in the CLOSING operation to obtain the same smearing effect. However, the most efficient and effective way to smear characters of x-height is to use a horizontal structuring element as described above.

A UNION of erosions is taken in step 314 of the image by using a set of ascender matching structuring elements such as those shown in FIGS. 4A–4F, and a set of descender matching structuring elements such as those shown in FIGS. 5A–5F. The UNION taken in step 314 provides optional noise elimination filtering, and the UNION will provide a seed from which to fill short image unit masks in a subsequent seed filling operation such as the fillClip operation of step 316. The UNION of step 314 acts on all image units remaining in the image (i.e., only short image units in this case) and since the UNION of erosions was taken using a set of ascender matching structuring elements and a set of descender matching structuring elements, the image units that will be filled are those containing ascender and/or descender characters, i.e., function words. The function words are identified in step 318 as those image units which are filled short image unit masks.

In step 320, a test occurs to determine whether a last page of the document has been scanned. If the last page has been scanned, then the method terminates at step 324, otherwise the page is incremented in step 322 and the incremented (next) page is scanned in step 302 whereupon the image (next page) is scanned and the previously described steps of the method are reiterated. Of course, all pages could be scanned and stored as bit map images in a memory prior to performing the function word identification procedures described above. Moreover, the image segmentation step can also be performed prior to performing this method and the segmented image stored in memory.

This is only one preferred method to perform the discrimination step 30 of FIG. 1. Using this method, the image units which have insufficient information content to be useful in evaluating the subject matter content of the document being processed are identified.

Next, in step 40, selected image units, e.g., the image units not discriminated in step 30, are evaluated, without decoding the image units being classified or reference to decoded image data, based on an evaluation of predetermined morphological (structural) image characteristics of the image units. The evaluation entails a determination (step 41) of the image characteristics and a comparison (step 42) of the determined image characteristics for each image unit with the determined image characteristics of the other image units.

One preferred method for defining the image unit image characteristics to be evaluated is to use the word shape derivation techniques disclosed in the copending U.S. patent application Ser. No. 07/794,391 filed concurrently herewith by D. Huttenlocher and M. Hopcroft, and entitled "A Method of Deriving Wordshapes for Subsequent Comparison," Published European Application No. 0543594, published May 26, 1993. As described in the aforesaid application, at least one, one-dimensional signal characterizing the shape of a word unit is derived; or an image function is derived defining a boundary enclosing the word unit, and the image function is augmented so that an edge function representing edges of the character string detected within the boundary is defined over its entire domain by a single independent variable within the closed boundary, without individually detecting and/or identifying the character or characters making up the word unit.

More specifically, the above reference discloses a method for deriving, defining, and comparing words in terms of their shapes. It will, of course, be recognized that each element of the system may be many devices, or may simply be a program operated within a single device. The method will be described with reference to FIG. 7. Beginning with an input bitmap 710, a bitmap of an image is initially directed to a segmentation system 712, in which words, or character strings, or other multi-character units of understanding, will be derived. Initially, the image bitmap passes through skew detector 714, which determines the angle of orientation of text in the image. Using information about the orientation of the image, and the image itself, at text baseline processor 716, toplines and baselines of the text are determined, so that upper and lower boundaries of lines of text within the image are identified.

At median filter 718, the function referred to as "blobify" is performed, which operates on the image so that each word group in a line may be treated as a single unit. As used herein, "word", "symbol string" or "character string" refers to a set of connected alphanumeric or punctuation elements, or more broadly, signs or symbols which together form a single unit of semantic understanding. It will be appreciated that these terms may also be used to refer to the images thereof. Such single units of understanding are characterized in an image as separated by a spacing greater than that which separates the elements, signs or symbols forming the unit. To the blobified image, a set of white lines are added at block 720, to clearly separate adjacent lines of text. The white lines are based on baseline determinations provided by processor 716. Using this information, i.e., the blobified words, which are clearly separated from adjacent words and words in adjacent lines, a bounding box is defined about the word at block 722, thereby identifying and enclosing the word.

Thereafter word shape signal computer 724 derives a word shape signal representing the individual words in the image, based on the original image and the bounding box determinations. This information is then available for use at a word shape comparator 726, for comparing word shape signals representative of known words from a word shape dictionary 728, with the as yet unidentified word shape signals. In an alternative embodiment word shape comparator 726 may be used to compare two or more word shapes determined from image 710. More importantly, word shape comparator 726 is not limited to the comparison of word shapes from unrecognized strings of characters to known word shapes. In a simplified context, comparator 726 is merely an apparatus for comparing one word shape against another to produce a relative indication of the degree of similarity between the two shapes.

In general, a method accomplishing this technique includes the following steps. Once orientation of the image is established and line spacing and word group spacing is established, each word can be surrounded by a bounding box. A reference line is then created extending through the character string image. The reference line may be a block having a finite thickness ranging from two-thirds of the x height to one-third of the x height, or in fact it may have a zero width. At the resolution of the image, the distance from the reference line to the upper edge of the text contour or bounding box is measured in a direction perpendicular to the reference line. Similarly, measurements may be made from the reference line to the lower bounding box edge or to the

text contour along the lower portion of the word, whichever is closer. Because the set of values derived computationally can be expressed in terms of position along the horizontal axis versus length, the signal can be considered a single independent variable or one dimensional signal. Either or both of these sets of values may be used to describe the word shape. Additionally, although possibly less desirable, it is well within the scope of this method to measure the distance of a perpendicular line drawn from the top of the bounding box or the bottom of the bounding box, to the first contact with the word or the reference line, as desired.

With a system and process for word shape derivation given, the method may also be considered mathematically. Considering image data i(x,y), which in one common case could be an array of image data in the form of a bitmap, a character set is identified in one of many methods, perhaps as described above, which defines a boundary enclosing the selected symbol string within a subset of the array of image data. From i(x,y), an edge signal e(x,y), which represents the edges of i(x,y) detected within the closed boundary, is derived. The edge signal is augmented by adding additional data to i(x,y) so that e(x,y) is a signal e'(x,y) defined over its entire domain with respect to a single dimension or variable within the closed boundary. One, two, or more signals may be derived from e'(x,y) which are each one dimensional signals g'(t), where g is a function of parameter t which is a reference frame dependent parameter.

It is important to realize that the mathematical process used for the derivation of the one dimensional signal is essentially reversible up to the information it contains, e.g., a bitmap may be reconstructed from the upper and lower bitmap contours. It will be noted that if the reference has a finite thickness and is therefore taken out of the image, that portion of the image is not identifiable, however, if it has a zero width the information still remains.

A recognition dictionary, or look up table of word shapes, can clearly be created through use of the described process. The process can be operated on using either scanned words as the source of the information, or in fact, they can be computer generated for a more "perfect" dictionary.

A detailed example using this method is disclosed in the U.S. patent application Ser. No. 07/794,391.

To demonstrate the process of the invention, at FIG. **10**, a sample image, taken from a public domain source is shown, having several lines of text contained therein. FIG. **10** demonstrates approximately how the image would appear on the page of text, while FIG. **11**, shows a scanned image of the page, which demonstrates an enlargement of the image of a bitmap that would present problems to known OCR methods. Looking at, for example, the image of the word **50***a* "practitioner" in the first line of the text image, it may be seen that several of the letters run together. Also, at the lower right hand portion of the image, circled and numbered **52**, noise is present. Looking at the word "practitioner's", circled and numbered **54**, the running together of a punctuation mark and a letter is further noted.

With reference again to FIG. **7**, in one possible embodiment of the invention, skew detector **714**, may be implemented using a general method for determining the orientation of the text lines in the image. This method looks at a small number of randomly selected edge pixels (defined as a black pixel adjacent to at least one white pixel), and for each edge pixel considers, at FIG. **12**A, a number of lines, **56***a*, **56***b*, **56***c* being examples, extending from the pixel at evenly spaced angular increments over a specified range of angles. The edge pixels are selected randomly from the set of all image pixels by the function RandomEdgePixel() (Appendix, page 243). FIGS. **12**A (see lines **56***a*, **56***b*, **56***c*), **12**B (see lines **58***a*, **58***b*, **58***c*) and **12**C (see lines **60***a*, **60***b*, **60***c*) represent a series of increasingly smaller angular ranges over which the above mentioned technique is applied to illustrative edge pixels to accurately determine the angular orientation of the text within the image. Subsequent to finding edge pixels and defining the lines, skew detector **714** traces the path of each line, determining the lengths, in pixels, of strings of successive black pixels which are intersected by the line. Upon reaching the image boundary, an average black pixel string length is calculated by summing the lengths of the individual strings, and dividing the sum by the total number of distinct strings which were found. This operation is carried out for all the lines, thereby arriving at an average black pixel string length for each line extending from the selected edge pixel. These lengths are plotted on FIG. **12**D as curve A, showing minima at approximately 0 and 3.14 radians. Curve A is a graphical representation of the summation/averaging function over each of a series of angled lines extending from the edge pixel, and spread over a range from 0 to 2π radians. Once a first minimum has been located, verification of the minimum (in the example, approximately 0 radians) is achieved by determining whether a second minimum exists at approximately π radians from the first minimum. Upon verifying the existence of a second minima (in the example, approximately 3.14 or π radians), a coarse skew angle is identified. Subsequently, it is necessary to more closely determine the skew angle of the text. This is accomplished by utilizing a number of lines which extend from a randomly selected edge pixel, where the lines differ by smaller angular increments, and the angular range is centered about the coarse skew angle. However, the fine skew angle may be determined by analyzing the total number of black pixels contained along a predetermined length of the lines. More specifically, the number of pixels over a unit distance are plotted as curve B on FIG. **12**D, and the fine skew angle is determined by identifying the maxima of the curve. In other words, the point of the curve where the highest concentration of black pixels per unit line length exists, more accurately represents the angle of the text lines in the image. As shown by curve B, this results in a fine skew angle of approximately 0 radians, where the line intersects with the most black pixels along its length, and therefore is representative of the closest angle of orientation that needs to be determined.

Alternatively, the skew angle may be determined as indicated by the NewFine() function (Appendix, page 245), which determines the skew angle using multiple iterations of the procedure described with respect to the fine angle determination. As indicated by FIGS. **12**A, **12**B, and **12**C, each iteration would also use lines covering an increasingly smaller angular range, until a desired skew angle accuracy is reached. In the implementation illustrated by FIGS. **12**A, **12**B, and **12**C, the desired accuracy is achieved by a series of three iterations, each using a series of 180 distinct angles about the selected edge pixel.

In the next process step, illustrated in the graphs of FIG. **13**A and FIG. **13**B, text baseline processor **716** identifies the characteristic lines, upper topline and lower baseline, of each line of text. The process steps executed by text baseline processor **716** are illustrated in detail in FIGS. **14**A and **14**B. The histogram of FIG. **13**A, shown to the left along the image, is derived by examining lines, at the resolution of the image, and oriented parallel to the skew orientation of the image, as defined by the previously determined skew angle. These parallel lines spanning the image are used to deter-

mine the number of black pixels intersected by each of the lines. Along lines passing through inter text line spaces, no black pixels should be intercepted, while along lines through the text, large numbers of black pixels should be intercepted.

More specifically, the function BaseLines(), (Appendix page 160), first finds the coordinates of a "main" line, block **142**, constructed through the center of the image and perpendicular to the text lines, as determined by the skew angle passed to the function as shown by block **140**. Next, Line Engine Procedure **144** is executed, where by proceeding along the main line from one end to the other, at a series of points along the main line, perpendicular branch lines are constructed which extend outwardly from the main line for a fixed distance, block **146**. Along the branch lines, the number of black vertical edge pixels are counted, block **148**, and the number of black pixels intersected by the lines are counted, block **150**, and summed for the opposing pairs of lines, block **152**. Black vertical edge pixels, as counted by block **148**, are defined as black pixels having a white neighboring pixel at either the upper or lower neighboring pixel position. LineEngine()procedure **144** is repeated until all points, and associated branch lines, along the main line have been processed, as determined by decision block **154**. An x-height value may be returned from this procedure, which will subsequently be used by the word shape computer **724**.

Subsequently, the counts for all the branch lines are analyzed to determine the branch line pairs having the highest ratio of black vertical edge pixels to black pixels. In general, those lines having the highest percentages would correspond to lines passing along the upper and lower edges of the characters which form the text lines. As illustrated in the enlarged view of FIG. **13B**, a definite distinction exists between those branch lines having a high vertical edge pixel ratio, line 82, and those having a low ratio, line 84. Application of a filter mask and comparison of the maximum peaks within the mask enables the identification of those lines which represent the text toplines and baselines, for example, line 82. The process is implemented in the maxFilter.c module, beginning at line 57, the code for which is also incorporated in the newBaselines.c module at line 274, page 214. Baseline determination is described in further detail in a copending U.S. patent application, for a "Method for Determining Boundaries of Words in Text", Huttenlocher et al., U.S. patent application Ser. No. 07/794,392, which has been previously incorporated herein by reference. An additional test may also be applied to the histogram operation of step 150. This added test, a boolean test, may be used to assure that a minimum run of black pixels was detected during the analysis of the line. For example, a flag, which is cleared at the start of each branch line analysis, may be set whenever a series of five sequential black pixels are detected along the line. This test would assure that small noise or image artifacts are not recognized as baselines due to a high vertical edge pixel ratio.

As an alternative method, it is possible to utilize the total number of black pixels lying along the branch lines to determine the locations of the baselines. Using histogram curve BL, which represents the number of black pixels counted along the branch lines, it is possible to determine which branch lines have the most black pixel intersections. Applying a threshold of the maximum allows the determination of the upper and lower characteristic line pairs for each text line. Hence, the rising and falling portions of the histogram curve BL, constitute the characteristic lines of the text, and the threshold would be used to specifically identify the localized maxima surrounding an intervening minima, thereby enabling identification of the baseline positions which would be used for further processing. More importantly, this alternative approach, illustrated as step 162, may be utilized to identify the upper and lower baselines of a baseline pair, based upon the slope of the BL histogram curve. It is important to note that there is little additional processing associated with the identification step as the histogram information was collected previously during step 150. Once the preliminary characteristic line or baseline pairs are identified, block **162**, a verification step, block **164**, is executed to verify that the baseline pairs are separated by more than a minimum distance, the minimum distance being established by calculating the average line pair separation for all line pairs in the image. After verification, the valid baseline information is stored by output block **166** for later use by the white line addition and segmentation blocks, **18** and **720**, respectively.

An important advantage of these baseline determination methods, are that they are highly insensitive to noise or extraneous marks in the interline space. FIG. **15** shows the result of the baseline determination on the example image of the sample text, showing that baseline pair, baseline and topline B_n and B_n', respectively, have been located on the image, indicating those portions of the image in which a predominant portion of the text occurs. While some portions of the character ascender strokes are outside the baselines, no detriment to the remainder of the process is noted. Of course, a smaller threshold value might enable the system to capture more of the ascending strokes.

With reference again to FIG. **7** in conjunction with FIGS. **16** and **17**, the next process step is a word group isolation step. A filter **718** is applied to a copy of the image which results in an image that tends to render the word into blobs distinguishable from one another. The filter is applied with a small window, to each area, to render as black those areas that are partly black. As shown in FIG. **16**, the blobify function (Appendix page 165) first initializes mask variables which establish the mask size and angle, block **180**, and then processes the upper scanline to initialize the data array, block **182**. Median filtering is accomplished by sequentially moving the mask window through the image, blocks **184** and **186**, and whenever the number of black pixels appearing in the window exceeds a threshold value, the target pixel, about which the window is located, is set to black. FIG. **17**, which illustrates some examples of the filter process, has a mask window **200** placed over a portion of the image. For example, with a twenty percent threshold and a generally rectangular mask having twenty-one pixels, arranged at an angel approximately equal to the skew determined for the text, the result of filtering in window **200** would be the setting of pixel **204** to black. Similarly, window **206**, which primarily lies within the intercharacter spacing between the pixel representations of the letters "r" and "o", would cause pixel **208** to be set to black. On the other hand, window **210**, which lies in the region between word groups, would not have a sufficient number of black pixels present within the window to cause pixel **212** to be set to black. The size, shape and orientation of mask window **200** is optimized to reduce the filling in between text lines, while maximizing the fill between letters common to a single word.

As illustrated by FIG. **18**, the result of the median filtering is that the relatively small spacing between characters in a word generally becomes inconsequential, and is filled with black pixels. Words become a single connected set of pixels, i.e., no white spaces completely separate characters in a single word. However, the relatively large spacing between character strings or between words, is a larger space outside

of the ability of the filter to turn into black, and therefore serves to distinguish adjacent symbol strings. With reference now to FIGS. **15** and **18**, it can be seen that the first two words of the sample text, "A" and "practitioner" have been "blobified", as this process is referred to, so that, for example, the "p" of "practitioner" is no longer separated from the "r" of that word. (Compare, FIG. **11**). Once again, despite the blobifying or blurring of characters, "A" and "practitioner" remain as discrete blobs of connected symbols, or words.

With reference again to FIG. **7**, as an adjunct to this step, white line addition **720**, superimposes upon the blobified image of FIG. **12** a series of white pixel lines to make certain that lines of text are maintained separately from adjacent lines of text (i.e., no overlapping of the filtered text lines). With reference to FIGS. **18** and **19**, noting the circled areas **258** and **258'**, a combination of an ascender and descender has resulted in an interline merging of two words. The text line overlap illustrated in area **258** of FIG. **18** is exactly what is eliminated by superimposing the white lines on the blobified or filtered image.

This superposition of white lines operation, the outcome of which is illustrated by FIG. **19**, is carried out by the process illustrated in FIG. **20** as executed in the DrawMiddleLines() function (Appendix page 233). Generally, white lines WL are added to the image, approximately halfway between adjacent baseline and topline pairs, to assure that there is no cross-text line blobifying. Once again, FIG. **19** shows the result of white line addition to the blobified image of FIG. **18**.

Referring now to FIG. **20**, white line addition block **720** begins by initializing variables in step 280 and subsequently reads in the topline location from the baseline information of the first text line. The topline information is discarded, block **282**, and the next baseline and topline locations are popped from the storage stack or list, blocks **284** and **286**, respectively. With respect to the image, this baseline-topline pair respectively represents the bottom and top of adjacent text lines. Next, at step 288, the point lying at the center of the pair is located to provide a starting point for the white lines which are drawn from the center of the image in an outward direction. The endpoints of the white lines are calculated in step 290, using the skew angle determined by skew detector **714** of FIG. **7**. White lines are drawn or superimposed on the blobified image at step 292, and the process is continued until all text lines have been effectively separated, as controlled by test block **294**.

With reference again to FIG. **7**, as a result of the blobify or median filtering, the position of bounding boxes about each connected set of pixels formed in the blobify step may be determined. Bounding boxes are placed only about those connected components or words that are in a text line lying between the superimposed white lines. The bounding boxes are placed at the orientation of the text line, by identifying the extreme points of each group of connected pixels in the direction of the text line, and in the direction orthogonal to the text line, as opposed to the image coordinate system. This operation is performed by the function FindBorders(), (Appendix, page 172). Generally, the FindBorders function steps through all pixels within the image to find the bounding boxes of the connected characters (Paint Component), to determine the coordinates of the upper left corner of each box, as well as the length and width of the box.

Referring now to FIGS. **21A** and **21B**, which detail the FindBorders() procedure, segmentation step 1022 begins by placing a white border completely around the filtered image, step 1300. This is done to avoid running outside the edge of the array of image pixels. Next, pixel and line counters, x and y, respectively, are initialized to the first pixel location inside the border. Calling the ReadPixel procedure, block **1304**, the pixel color (black or white) is returned and tested in block **1306**. If the pixel is white, no further processing is necessary and processing would continue at block **1322**. Otherwise, the PaintComponent() procedure (Appendix, page 171) is called and begins by storing the location of the black pixel in a queue, block **1308**. Subsequently, in a copy of the image, the pixel is set to white and the boundaries of the box, surrounding the connected pixels or components, are updated, blocks **1310** and **1312**, respectively. Next, adjoining black pixels are set to white, block **1314**, and the locations of the black pixels are added to the end of the queue, block **1316**. At block **1318** the queue pointers are tested to determine if the queue is empty. If not empty, the next pixel in the queue is retrieved, block **1320**, and processing continues at block **1312**. Otherwise, if the queue is empty, all of the connected black pixels will have been set to white and the box boundaries will reflect a box which encompasses the connected components. Subsequently, the boundaries of the box which encompasses the word segment are verified and may be adjusted to an orthogonal coordinate system oriented with respect to the skew of the text lines, block **1322**.

It will no doubt be apparent here that while finding each text line is an integral part of the described method, and serves to make the present embodiment more robust, other methods of deriving the information acquired by that step are possible. The primary use of the text line finding function is a) to determine x-height, and b) define the white line addition for separating interline blobs. Certainly this step may be removed, with a sacrifice in robustness, or other means of deriving the necessary information may be available.

The looping process continues at block **1324** which checks pixel counter x to determine if the end of the scanline has been reached, and if not, increments the counter at block **1326** before continuing the process at block **1304**. If the end of the scanline has been reached, pixel counter x is reset and scanline counter y is incremented at block **1328**. Subsequently, block **1330** checks the value of scanline counter y to determine if the entire image has been processed. If so, processing is completed. Otherwise, processing continues at block **1304** for the first pixel in the new scanline.

Thus, as shown in FIG. **22**, for the word "practitioner" the extremities of the connected character image define the bounding box. Once bounding boxes have been established, it is then possible at this step, to eliminate noise marks from further consideration. Noise marks are determined: 1) if a bounding box corner is outside the array of image pixels (Appendix, page 171); 2) if a box spans multiple text lines in the array (Appendix 229), or lies completely outside a text line; 3) if boxes are too small compared to a reference ϵ, in either or both longitudinal or latitudinal directions, and accordingly are discarded. Noise marks **70a** and **72** and others will not be considered words. The OnABaseline() function (Appendix, page 229) is an example of a function used to eliminate those boxes lying outside of the baseline boundaries.

With reference to FIG. **7**, at word shape computer **724**, a signal representing the image of a word, or at least a portion thereof, now isolated from its neighbors, is derived. The derived signal is referred to as a word shape contour. The shape contour for each word is determined using the MakeShell() function (Appendix, page 228). As illustrated in FIG. **23A**, this function first moves along the top of each bounding box, and starting with each pixel location along the top of the box, scans downward relative to the page

orientation, until either a black pixel, or the bottom of the box, is reached. A record of the set of distances d between the top of the box and the black pixel or box bottom is maintained. The set of distances d, accumulated over the length of the box, constitutes the top raw contour of the word shape. Subsequently, a bottom raw contour is produced in a similar manner as illustrated in FIG. 23B, for the same word depicted in FIG. 23A, by sequentially moving across the bottom of the box, and looking in an upwards direction, for either the first black pixel or the top of the bounding box.

With reference now to FIG. 25, at block 100 which preferably operates on the actual image as opposed to the filtered image, which could be used in this step, one or more reference lines are established through each word. In other terms, the data representing the symbol string is augmented, so that it is defined over the range of the symbol string. In one embodiment, a blackout bar, which may have a finite thickness or a zero thickness is constructed through the word, preferably having an upper limit or reference line at approximately two thirds of the x height, and a lower limit or reference line at approximately one-third of the x height (which was determined at the baseline determination step). At contour calculation 102, a set of measurements is derived, for the distance d between the upper or lower edge of the bounding box, and the word, or the nearer of the reference line's closer edge of the black out bar. The calculation's measurements are made at the resolution of the image. With reference to FIG. 26A, where the calculation's measurements are illustrated pictorially, it can be seen that the reference lines serve to allow the signal that will ultimately be derived from this step to be defined at every sampling position over the length of the word. In a preferred embodiment, the calculation's measurements of d are actually generated from the contour data derived in accordance with FIGS. 23A, 23B previously collected, and are adjusted to limit the distance d with either the upper or lower edge of the blackout bar as indicated. In the embodiment shown in FIG. 26A, measurements are made from the upper line of the bounding box to the upper reference line of the word, although this is not a requirement. Thus, for example, the measurement could alternatively be made from the reference line to either the upper or lower bounding line, or the character. FIG. 26B better shows how the set of measurements is used to form the signal output from block 104. The contour is represented as a set of measurements distance d', relative to the reference line. Measurement d' is therefore derived from the measurements shown in FIG. 26A, which designate the stopping point of line d, and the known position of the black out bar. Calculating the distance relative to the reference line enables scaling of the word shape contours to a common x height, thereby facilitating any subsequent comparison of the shapes. Accordingly, the distances d' represent a measurement from the reference line or blackout bar to the outer limits of the letter, and in the absence of a letter, provide a zero measurement. These measurement might be derived directly, but the proposed indirect methods appear easier to implement. FIGS. 26C and 26D show that the sets of d' values can be plotted on a graph to form a one dimensional signal or curve representing the word shape. Details of the contour determination are contained in the function StoreOutlinePair() beginning in the Appendix at page 255. FIG. 24 is an image of the contour locations as established for the text sample of FIG. 10. It is important to note the informational content of FIG. 24, where, for the most part, it is relatively easy to recognize the words within the passage by their contours alone.

In studies of the information delivered by the appearance of English language words, it has been determined that in a majority of cases, words can be identified by viewing only approximately the top third of the image of the word. In other words, the upper portion of the word carries with it much of the information needed for identification thereof. In a significant portion of the remainder of cases, words that are unidentifiable by only the upper third of the image of the word, become identifiable when the identification effort includes the information carried by the lower third of the image of the word. A relatively small class of words requires information about the middle third of the word before identification can be made. It can thus be seen that a stepwise process might be used, which first will derive the upper word shape signal or contour, second will derive the lower word shape signal or contour, and thirdly derive a word shape signal central contour (from the reference line towards the word or bounding box), in a prioritized examination of word shape, as required. In the examples of FIG. 26A, 26B, and 26C, the word "from" is fairly uniquely identifiable from its upper portion only. In the examples of FIG. 27A, 27B, 27C and 27D, the word "red" is less uniquely identifiable from its upper portion, since it may be easily confused with the word "rod", and perhaps the word "rad". While the lower portion of the letter "a" may distinguish "red" and "tad", it is doubtful that the lower portion of the letter "o" will distinguish the words "red" from "rod". However, the central portions of "red", "rad", and "rod" are quite distinct.

The determined morphological image characteristic(s) or derived image unit shape representations of each selected image unit are compared, as noted above (step 42), either with the determined morphological image characteristic(s) or derived image unit shape representations of the other selected image units (step 42A), or with predetermined/user-selected image characteristics to locate specific types of image units (step 42B). The determined morphological image characteristics of the selected image units are advantageously compared with each other for the purpose of identifying equivalence classes of image units such that each equivalence class contains most or all of the instances of a given image unit in the document, and the relative frequencies with which image units occur in a document can be determined, as is set forth more fully in the copending U.S. patent application Ser. No. 07/795,173 filed concurrently herewith by Cass et al., and entitled "Method and Apparatus for Determining the Frequency of Words in a Document without Document Image Decoding." Image units can then be classified or identified as significant according the frequency of their occurrence, as well as other characteristics of the image units, such as their length. For example, it has been recognized that a useful combination of selection criteria for business communications written in English is to select the medium frequency word units.

The method for determining the frequency of words without decoding the document is shown in FIG. 8. The image is segmented into undecoded information containing image units (step 820) by using the method described above or by finding word boxes. Word boxes are found by closing the image with a horizontal SE that joins characters but not words, followed by an operation that labels the bounding boxes of the connected image components (which in this case are words). The process can be greatly accelerated by using 1 or more threshold reductions (with threshold value 1), that have the effect both of reducing the image and of closing the spacing between the characters. The threshold reduction(s) are typically followed by a closing with a small horizontal SE. The connected component labeling operation

is also done at the reduced scale, and the results are scaled up to full size. The disadvantage of operating at reduced scale is that the word bounding boxes are only approximate; however, for many applications the accuracy is sufficient. The described method works fairly well for arbitrary text fonts, but in extreme cases, such as large fixed width fonts that have large inter-character separation or small variable width fonts that have small inter-word separation, mistakes can occur. The most robust method chooses a SE for closing based on a measurement of specific image characteristics. This requires adding the following two steps:

(1) Order the image components in the original or reduced (but not closed) image in line order, left to right and top to bottom.

(2) Build a histogram of the horizontal inter-component spacing. This histogram should naturally divide into the small inter-character spacing and the larger inter-word spacings. Then use the valley between these peaks to determine the size of SE to use for closing the image to merge characters but not join words.

After the bounding boxes or word boxes have been determined, locations of and spatial relationships between the image units on a page are determined (step 825). For example, an English language document image can be segmented into word image units based on the relative difference in spacing between characters within a word and the spacing between words. Sentence and paragraph boundaries can be similarly ascertained. Additional region segmentation image analysis can be performed to generate a physical document structure description that divides page images into labelled regions corresponding to auxiliary document elements like figures, tables, footnotes and the like. Figure regions can be distinguished from text regions based on the relative lack of image units arranged in a line within the region, for example. Using this segmentation, knowledge of how the documents being processed are arranged (e.g., left-to-right, top-to-bottom), and, optionally, other inputted information such as document style, a "reading order" sequence for word images can also be generated. The term "image unit" is thus used herein to denote an identifiable segment of an image such as a number, character, glyph, symbol, word, phrase or other unit that can be reliably extracted.

Advantageously, for purposes of document review and evaluation, the document image is segmented into sets of signs, symbols or other elements, such as words, which together form a single unit of understanding. Such single units of understanding are generally characterized in an image as being separated by a spacing greater than that which separates the elements forming a unit, or by some predetermined graphical emphasis, such as, for example, a surrounding box image or other graphical separator, which distinguishes one or more image units from other image units in the document image. Such image units representing single units of understanding will be referred to hereinafter as "word units."

A discrimination step 830 is next performed to identify the image units which have insufficient information content to be useful in evaluating the subject matter content of the document being processed by using the technique described above.

Next, in step 840, selected image units, e.g., the image units not discriminated in step 830, are evaluated, without decoding the image units being classified or reference to decoded image data, based on an evaluation of predetermined image characteristics of the image units. The evaluation entails a determination (step 841) of the image characteristics and a comparison (step 842) of the determined image characteristics for each image unit with the determined image characteristics of the other image units.

One preferred method for defining the image unit morphological image characteristics to be evaluated is to use the word shape derivation techniques previously discussed. At least one, one-dimensional signal characterizing the shape of a word unit is derived; or an image function is derived defining a boundary enclosing the word unit, and the image function is augmented so that an edge function representing edges of the character string detected within the boundary is defined over its entire domain by a single independent variable within the closed boundary, without individually detecting and/or identifying the character or characters making up the word unit.

The determined image characteristic(s), e.g., the derived image unit shape representations of each selected image unit are compared, as noted above (step 841), with the determined image characteristic(s)/derived image unit shape representations of the other selected image units for the purpose of identifying equivalence classes of image units (step 850), such that each equivalence class contains most or all of the instances of a given word in the document. The equivalence classes are thus formed by clustering the image units in the document based on the similarity of image unit classifiers, without actually decoding the contents of the image units, such as by conversion of the word images to character codes or other higher-level interpretation. Any of a number of different methods of comparison can be used. One technique that can be used, for example, is by correlating the raster images of the extracted image units using decision networks, such technique being described for characters in a Research Report entitled "Unsupervised Construction of Decision networks for Pattern Classification" by Casey et al., IBM Research Report, 1984, herein incorporated in its entirety.

Depending on the particular application, and the relative importance of processing speed versus accuracy, for example, comparisons of different degrees of precision can be performed. For example, useful comparisons can be based on length, width or some other measurement dimension of the image unit (or derived image unit shape representation e.g., the largest figure in a document image); the location of the image unit in the document (including any selected figure or paragraph of a document image, e.g., headings, initial figures, one or more paragraphs or figures), font, typeface, cross-section (a cross-section being a sequence of pixels of similar state in an image unit); the number of ascenders; the number of descenders; the average pixel density; the length of a top line contour, including peaks and troughs; the length of a base contour, including peaks and troughs; and combinations of such classifiers.

One way in which the image units can be conveniently compared and classified into equivalence classes is by comparing each image unit or image unit shape representation when it is formed with previously processed image units/shape representations, and if a match is obtained, the associated image unit is identified with the matching equivalence class. This can be done, for example, by providing a signal indicating a match and incrementing a counter or a register associated with the matching equivalence class. If the present image unit does not match with any previously processed image unit, then a new equivalence class is created for the present image unit.

Alternatively, as shown (step 50) the image units in each equivalence class can be linked together, and mapped to an equivalence class label that is determined for each equivalence class. The number of entries for each equivalence class can then be merely counted.

Thus, after the entire document image, or a portion of interest, has been processed, a number of equivalence classes will have been identified, each having an associated number indicting the number of times a image unit was identified having similar morphological characteristics, or classifiers, thus determining the image unit frequency.

It will also be appreciated that the selection process can be extended to phrases comprising identified significant image units and adjacent image units linked together in reading order sequence. The frequency of occurrence of such phrases can also be determined such that the portions of the source document which are selected for summarization correspond with phrases exceeding a predetermined frequency threshold, e.g., five occurrences. A preferred method for determining phrase frequency through image analysis without document decoding is disclosed in copending U.S. patent application Ser. No. 07/774,555 filed concurrently herewith by Withgott et al., and entitled "Method and Apparatus for Determining the Frequency of Phrases in a Document Without Document Image Decoding."

It will be appreciated that the specification of the image characteristics for titles, headings, captions, linguistic criteria or other significance indicating features of a document image can be predetermined and selected by the user to determine the selection criteria defining a "significant" image unit. For example, titles are usually set off above names or paragraphs in boldface or italic typeface, or are in larger font than the main text. A related convention for titles is the use of a special location on the page for information such as the main title or headers. Comparing the image characteristics of the selected image units of the document image for matches with the image characteristics associated with the selection criteria, or otherwise recognizing those image units having the specified image characteristics permits the significant image units to be readily identified without any document decoding.

Any of a number of different methods of comparison can be used. One technique that can be used, for example, is by correlating the raster images of the extracted image units using decision networks, such technique being described in a Research Report entitled "Unsupervised Construction of Decision networks for Pattern Classification" by Casey et al., IBM Research Report, 1984, herein incorporated in its entirety.

Preferred techniques that can be used to identify equivalence classes of word units are the word shape comparison techniques disclosed in U.S. patent application Ser. Nos. 07/796,119 and 07/795,169, filed concurrently herewith by Huttenlocher and Hopcroft, and by Huttenlocher, Hopcroft and Wayner, respectively, and entitled, respectively, "Optical Word Recognition By Examination of Word Shape," Published European Application No. 0543592, published May 26, 1993, and "Method for Comparing Word Shapes."

For example, U.S. patent application Ser. No. 07/795,169 discloses, with reference to FIG. 7, one manner in which a comparison is performed at word shape comparator **726**. In one embodiment, the comparison is actually several small steps, each of which will be described. With reference to FIG. **28**, generally, the two word shape signals, one a known word, the other for an unknown string of characters are compared to find out whether they are similar. However, in this case, signal R is the upper contour of the word "red", while signal F is the upper contour of the word "from". Actually, relatively few signals could be expected to be exactly identical, given typical distinctions between character fonts, reproduction methods, and scanned image quality. However, the word shape signals to be compared may be scaled with respect to one another, so that they have the same x-heights. This is achieved by determining the x-height of the pair of word shape contours to be compared. Once determined, the ratios of the x-heights are used to determine a scale factor to be applied to one of the contours. As the x-height is a characteristic measurement for fonts, it is used to determine the scaling factor in both the horizontal and vertical directions. An example of the scaling operation is found in the fontNorm.c file beginning at line 172, where the StoreOutlinePair() function carries out the scaling operation in both the x and y, horizontal and vertical, directions. Alternatively, the shape signals may be compared without normalization and a weighing factor imposed upon the portion of the measured difference due to the unequal lengths. Furthermore, the amplitude or height of the signals has been normalized to further reduce the impact of the font size on the word shape comparison.

Referring next to FIGS. **29A–29C**, which illustrate details of the ascender/descender normalization operation, each of the shape signals are normalized based upon a common relationship between the ascender and descender heights and the x-height of the text characters. As illustrated, the actual ascender heights of characters printed with supposedly similar font size, or what is now an appropriately scaled font size, may be slightly different. This occurs as a result of type faces or fonts which are small on body or large on body, implying that similar characters exhibit variations in height across fonts that are the same size, for example **24** point fonts. As an illustration, distance d_1 in FIG. **29A** represents the difference in ascender height for two occurrences of the letter "h." Likewise, distance d_2 illustrates a similar difference between the heights of the letter "f" in FIG. **29B**. As illustrated in FIG. **29C**, the typical character may be broken into three sections, ascender portion **390**, x-height portion **392**, and descender portion **394**. In addition, the relative heights of these sections are illustrated as c, a, and b, respectively. Again, the normalization operation applied to the shape contours is found in the fontNorm.c module, beginning at page 183 of the Appendix. Applying the operations described with respect to StoreOutlinePair() function, page 255 of the Appendix, the areas of the contour lying above the x-height are scaled as follows:

$$f(t)=\frac{1.5}{a+c}\cdot f(t)$$

Similarly, the descenders are scaled by the following equation:

$$f(t)=\frac{1.5}{a+b}\cdot f(t)$$

where, in both cases, the value used in the numerator (1.5) is arrived at based upon observation of the relationship between ascender or descender heights and the x-height. Also included within the StoreOutlinePair() function is an operation to remove the portions of the contours which do not represent portions of the text string. These regions lie at the ends of the bounding boxes illustrated in FIG. **22**. For example, the box surrounding the word "practitioner" in FIG. **22** can be seen to extend beyond the actual word image. As further illustrated at the ends of the word "from" in FIGS. **26A–26D**, the contour does not contain useful information. By removing these regions from the contour shape, less error will be introduced into the comparison operations.

Subsequent to the normalization operation, standard signal processing steps can be used to determine the similarity or dissimilarity of the two signals being compared. Alternatively, the following equation may be used:

$$\Delta_{string} = \sqrt{\int_0^1 (f(x) - g'(x))^2 dx}$$

where

Δ_{string} is the difference value between the two signals;

f(x) is the known signal; and

g'(x) is the unknown signal.

In a simple determination, the difference could be examined and if it is close to zero, such would be indicated that there would be almost no difference between the two signals. However, the greater the amount of difference, the more likely that the word was not the same as the word to which it was being compared.

It is important to note that the embodiments described herein, as supported by the code listings of the Appendix, compare the word shape contours using the upper and lower contours for each word in conjunction with one another. This is an implementation specific decision, and is not intended to limit the invention to comparisons using only the top and bottom contours in conjunction with one another. In fact, sufficient information may be contained within the upper contours alone so as to significantly reduce the requirements for a comparison of the lower contours, thereby saving considerable processing effort.

The steps of this simplified comparison method, as first contemplated, are illustrated in FIG. **30**. Beginning at step 410, the contour for the first word shape is retrieved from memory, and subsequently, the second word shape is retrieved by step 412. Next, the centers of gravity of the word shapes, defined by the upper and lower contours, are determined and aligned, step 414. The purpose of this step is to align the centers of the word contours to reduce the contour differences that would be attributable solely to any relative shift between the two sets of contours being compared. The center of gravity is determined by summing the areas under the curves (mass) and the distances between the contours (moments) which are then divided to give an indication of the center of gravity for the upper and lower contour pair. Once determined for both sets of contour pairs, the relative shift between the pairs is determined, step 416, and the contours are shifted prior to calculating the difference between the contours. The shifting of the contours is necessary to reduce any error associated with the establishment of the word shape boundaries and computation of the word shapes at block **724** of FIG. 7. Step 418 handles those regions lying outside the overlapping range of the shifted contour pairs, determining the difference against a zero amplitude signal in the non-overlapping regions. This is done by summing the squared values of the upper and lower contours at the non-overlapping ends of the contours. Subsequently, the overlapping region of the contours are compared, step 420. The difference in this region is determined as the sum of the squared differences between the upper curves and the lower curves, as shown in the function L2Norm() on page 100 of the Appendix. Next, the values returned from steps 418 and 420 are added to determine a sum of the differences over the complete range defined by the shifted contours. This value may then be used as a relative indication of the similarity between the contour pairs for the two word shapes being compared.

An alternative to the center-of-gravity comparison method, uses a signal processing function known as time warping, as described in the article "Performance Tradeoffs in Dynamic Time Warping Algorithms for Isolated Word Recognition", by Myers, Rabiner, and Rosenberg, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-28, No. 6, December 1980, and the book, "Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison", by Sankoff and Kruskal, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1983, Chapters 1 and 4, and may be used to provide for compression and expansion of points along the contours until the best match is made. Then a score is derived based on the amount of difference between the contours being compared and the stretching required to make the contours match. Once again, the score provides a relative indication of the match between the two signals being compared.

Referring now to FIG. **31**, which depicts the general steps of the dynamic warping method, the method relies on the use of a difference array or matrix to record the distances between each point of the first contour and points of the contour to which it is being compared. As illustrated in the figure, and detailed in the code listings contained in the Appendix, the process is similar for all of the measures which may be applied in the comparison.

First, the organization of the code is such that a data structure is used to dynamically control the operation of the various comparison functions. The structure DiffDescriptor, the declaration for which is found on page 9 of the Appendix (see diff.h), contains variables which define the measure to be applied to the contours, as well as, other factors that will be used to control the comparison. These factors include: normalization of the contour lengths before comparison; separate comparisons for the upper and lower contours; a centerWeight factor to direct the warping path; a bandwidth to constrain the warp path; a topToBottom ratio which enables the top contour comparison to be weighted more or less with respect to the bottom contour comparison; and a hillToValley ratio to selectively control weighing the contour differences when an unknown contour is being compared to a known or model word shape contour. Interpretation of the various factors is actually completed in the diff2.c module at page 56 of the Appendix, although descMain.c at page 49 provides an illustration of the interpretation of the factors.

In general, each measure implements a comparison technique, however, each is optimized for a specific type of dynamic comparison, for example, a slope limited dynamic warp having a non-unitary centerweight and a topToBottom weight greater than one. The first level of selection enables the use of a slope-constrained warping function for comparison, an unconstrained warp, or a simple, non-warped, comparison. Within both of the warp comparison methods, there are both separate comparison functions, where the top and bottom contours are warped independently, and parallel comparison functions, where the warp is applied to both the top and bottom contours simultaneously. Specific details of the comparison functions are generally contained within the newMatch.c file beginning at page 101 of the Appendix.

In the general embodiment, the dynamic warping process starts by allocating space for the path/distance array, step 450, which will hold the distance values generated during the comparison and warping of one word shape contour with respect to another. After allocating space, the border regions of the array must be initialized as the process used by all the warping measures is an iterative process using data previously stored in the array for the determination of the cumulative difference between the contours. At step 452, the array borders are initialized. Initialization of the first row of the array entails the determination of the square of the difference between a first point on the first contour and each point on the second contour. Subsequent to border initial-

ization, the column and row index values, L1 and L2, respectively, are reset to 1 to begin processing the individual, non-border, points along the contours.

Processing of the contours proceeds at steps 458 through 464, where the difference in distance between each point along the second contour, with respect to a point on the first contour is calculated. Moreover, this difference, or distance, is calculated and then summed with a previously determined difference value. In addition, some of the previously determined difference values may be weighted differently, for example, in one embodiment weights of the difference values along the array diagonal may be modified by a centerWeight weighing factor. As an illustration, the operation of the NewMatch() function, beginning at line 106 on page 103, at first, the distance (rest) is calculated as the sum of the squares of the differences between a point on the first contour and a point on the second contour, over the upper and lower contours, where the top contour difference is weighted by the topToBottom variable. This distance (rest) is used in subsequent iterations to determine the horizontal, vertical and diagonal difference values in the loop beginning at line 137 on page 103. To determine each of these values, the current distance value, represented by rest, would be added to the previous values in the down, left, and down-left array positions, the down-left position value being the diagonal position which is weighted by the centerWeight factor as previously described. Referring to FIG. **32A**, which illustrates the positional relationship between a previously determined value X, at array location **502**, and subsequent array locations, the value X might be added to the difference values of subsequent locations to accumulate the total difference. calculations is shown. When calculating the difference value for array location **504**, the value in location **502** would be used as the down value. Similarly, when calculating the value in location **506**, the value of location **502** would be used as the center-weighted down-left, or diagonal, value. After calculating the three difference values, steps 458, 460, and 462, the process continues by selecting the smallest of the three values, step 464, for insertion into the current array position, step 466. As illustrated in the Appendix at line 144 of page 103, the FMin() function from page 101 returns the minimum of the three values previously calculated, the value being inserted into the storage array pointed to by pointer dc.

Subsequently, the process illustrated in FIG. **31** continues by determining the differences between the point on the first contour, represented by L1, to points on the second contour, represented by L2. Decision step 468 controls the iterative processing of the points along the second contour by testing for the end of the contour, or swath. In the implementation shown in the Appendix, the index variables i and j are used in place of L1 and L2 to control the difference calculation loops. As indicated in the code for the NewMatch function beginning on page 102 of the Appendix, the swath is referred to as the bandwidth, and is determined by a desired bandwidth which is adjusted for the slope defined by the contour lengths (see page 102, lines 83–89). If no limit has been reached, processing for the next point would continue at step 458 after the value of L2 was incremented at step 470. Similarly, decision step 472 controls the processing of each point along the first contour, in conjunction with incrementing step 474. Once all the points have been processed with respect to one another, as evidenced by an affirmative response in step 472, the relative difference score, best score, is contained in the farthest diagonal position of the array (L1, L2). Subsequently, the value determined at step 476 is returned as an indication of the dynamically warped difference between the contours being compared.

The code implementation found in the NewMatch() function on page 103 of the Appendix has optimized the execution of the aforedescribed warping process by reducing the large two-dimensional array to a pair of linear arrays which are updated as necessary. Due to this modification, the minimum difference, or best score, for the warp comparison value is found in the last location of the one-dimensional array. Furthermore, the final difference value, dc, may be subsequently normalized to account for the length differences between the two sets of contours being compared. Finally, such a value might subsequently be compared against a threshold or a set of similarly obtained difference values to determine whether the contours are close enough to declare a match between the words, or to determine the best match from a series of word shape comparisons.

In yet another embodiment, the dynamic time warping process previously described may be altered to compare the difference values contained in the difference array to a threshold value on a periodic basis. Upon comparison, the process may be discontinued when it is determined that sufficient difference exists to determine that the contours being compared do not match one another, possibly saving valuable processing time. Moreover, the sequential operation of word shape comparator **726** might be done in conjunction with sequential output from word shape computer **724**, thereby enabling the parallel processing of a textual image when searching for a keyword.

Having described a basic implementation of the dynamic warping comparison measures, the distinctions of the other dynamic warp comparison methods included in the Appendix and the application of the control factors previously mentioned will be briefly described to illustrate the numerous possible embodiments of the present invention. First, the method previously described may also be implemented with the slope of the warp path being constrained as it moves across the array. Details of the implementation are found in the SlopeCMatch() function beginning on page 111 of the Appendix. This measure is further illustrated graphically in FIG. **32B**, where the value of array location **512**, X, may be added to only the three subsequent array locations shown. For example, X may be added to array location **514**, when considered as the **d2L1** value for location **514**. The nomenclature used for the variable names, and followed in the figure, is as follows: **d2L1** refers to the array location which is down 2 rows and left one column, **d1L1**, refers to the lower left diagonal array location, and **d1L2** refers to the array location that is down one column on left 2 rows from the current array location. In a similar manner, X may be added as the **d1L2** value for the calculation of the cumulative difference value for array location **516**.

As is apparent from a comparison of FIGS. **32A** and **32B**, the slope constrained warping measure limits the warping path which can be followed during the generation of the cumulative difference value. The reason for implementing such a constraint is to prevent the warping process from removing, or compressing, a large area of one of the two contours being compared, without imposing a significant "cost" to such a compression.

Next, the method previously described with respect to the parallel warping process may also be implemented on only one pair of contours at a time, for example, the upper contours of two word shapes. The functions SepMatch() and SepCMatch(), as found in the Appendix on pages 104 and 113, respectively, implement the separate matching measure in both the non-slope-constrained and slope-constrained fashions previously described. In general, these measures separately calculate the difference between the top or bottom

contours of a pair of wordshapes. The general implementation indicated for the measures in the code shows that these measures are typically used sequentially, first determining the warped difference for the top contours, and then adding to it the warped difference from the bottom contour comparison, resulting in a total difference for the wordshapes.

By carrying out the comparison methods described in a "piece-wise" cascaded fashion, further processing benefits may also be derived. More specifically, cascaded comparison would entail, first, utilizing the upper contours of the words being compared to identify a word, or at least narrow the set of possible alternatives and, second, using the lower contour comparison to provide complete identification. It is believed that such an approach to word shape comparison operation **726** would considerably reduce processing time spent on identifying unknown word shapes by comparison to a dictionary of known word shapes, **728**, as illustrated in FIG. 7. Important to the cascaded comparison, is the constraint that the top and bottom warps applied to the contours must be relatively equivalent. This requirement arises from the fact that the upper and lower curves have a relationship to a common word, and if this relationship is not maintained during the warp analysis, the accuracy of the comparison will be compromised.

Alternatively, the dynamic warping technique may be applied as described, with the addition of a function suitable for accumulating the relative warp applied to the upper and lower curves in achieving the best match. For example, when a known, non-italicized word shape is compared to an unknown word shape, a shift in the warp applied to the upper curve relative to the lower curve could be indicative of an italicized word, however, the length of the warped region will remain the same for the top and bottom warps. Such a technique may prove useful in the identification of important words within a larger body of text, as these words are occasionally italicized for emphasis.

One of the control factors which has not been previously described is the bandwidth factor. As implemented, the bandwidth factor controls the relative width of the signal band in which the warping signal will be constrained. More specifically, the band width limitation is implemented by defining a region about the array diagonal in which the warp path which traverses the array is constrained. The constraint is implemented by assigning large values to those areas outside of the band width, so as to make it highly unlikely that the path would exceed the constraint.

Another factor which was briefly mentioned is the topToBottom factor. When applied, the value of this variable is used to weight the difference value determined for the top contour warping process. Therefore, use of a number greater than one, will cause the upper contour difference to be weighted more heavily than the lower contour difference. A very large number would effectively eliminate the lower contour difference completely and, likewise, a zero value would eliminate the upper contour difference completely. This factor is generally considered important to enable the upper contour to be weighted in proportion to its information content, as it generally carries more information regarding the word than does the lower contour.

The hillToValley ratio is a variable which is usually applied in situations when a known, or model, set of word shape contours is being compared against a set of word shape contours from an unknown image. In exercising this option, the model set of contours is passed as the comparison measure functions, for example, NewMatch() on page 102 of the Appendix. When determining the difference between points on the contours, the comparison functions commonly call the function SquareDifference() on page 101 of the Appendix to determine the sum of the squared difference. SquareDifference() applies the hillToValley ratio to the squared difference whenever it determines that the value of the model contour is less than the contour being compared. The result of applying a hillToValley value greater than one is that the relative "cost" of the difference when the model contour is less than the target contour is smaller than the same difference when the model contour is greater than the target contour. The basis for this type of weighing is that when comparing against a model contour, the comparison should treat those areas of the target contour that are subject to being "filled in" during a scanning or similar digitizing operation with less weight than regions not likely to be filled in, as evidenced by contour positions below the model contour. For instance, the regions where ascenders and descenders meet the body of the character are likely to be filled in during scanning, thereby causing the target contour to have a gradual contour in those regions, whereas the model contour would most likely have a defined peak or valley in these regions. Hence, the contour value of the model would be less than the contour value of the target, even though the characters may have been identical. Therefore, the hillToValley variable attempts to minimize the impact to the calculated difference value over these regions.

It is important to note that the aforedescribed measures and control factors allow the comparison measures to be conducted in numerous permutations. However, the flexibility which these measures permit is intended to enhance the applicability of the comparison process, so that when information is known about a particular word shape contour, for example, a model contour generated from a computer generated character font, the measures may place reliance on that information to make the comparisons more robust.

The mathematical explanation of the word shape derivation process suggests that alternative methods of deriving the word shape signal exist. Some possible alternatives are the establishment of the one dimensional signal using an alternative coordinate scheme, for example polar coordinates. Another possibility is generation of signal g(t), where g(t) represents the direction from each contour point to the succeeding contour point, where t would represent the point number.

Depending on the particular application, and the relative importance of processing speed versus accuracy, for example, comparisons of different degrees of precision can be performed. For example, useful comparisons can be based on length, width or some other measurement dimension of the image unit (or derived image unit shape representation, e.g., the largest figure in a document image); the location or region of the image unit in the document (including any selected figure or paragraph of a document image, e.g., headings, initial figures, one or more paragraphs or figures), font, typeface, cross-section (a cross-section being a sequence of pixels of similar state in an image unit); the number of ascenders; the number of descenders; the average pixel density; the length of a top line contour, including peaks and troughs; the length of a base contour, including peaks and troughs; the location of image units with respect to neighboring image units; vertical position; horizontal inter-image unit spacing; and combinations of such classifiers. Thus, for example, if a selection criteria is chosen to produce a document summary from titles in the document, only title information in the document need be retrieved by the image analysis processes described above. On the other hand, if a more comprehensive evaluation of the document contents is desired, then more comprehensive identification techniques would need to be employed.

In addition, morphological image recognition techniques such as those disclosed in concurrently filed U.S. patent application Ser. No. 07/775,174, to Bloomberg et al., and entitled "Methods and Apparatus for Automatic Modification of Selected Semantically Significant Portions of a Document Without Document Image Decoding", can be used to recognize specialized fonts and typefaces within the document image.

More particularly, the above reference provides a method for automatically emphasizing selected information within the data or text of a document image. Referring to FIG. 9, the first phase of the image processing technique of the method involves the segmentation of the image into undecoded information containing image units (step 920) using techniques described above. Then the locations of and spatial relationships between the image units on a page is determined (step 925), which was previously described.

The discrimination step 930, which was previously described, is next performed to identify the image units which have insufficient information content to be useful in evaluating the subject matter content of the document being processed. Such image units include stop or function words, i.e., prepositions, articles and other words that play a largely grammatical role, as opposed to nouns and verbs that convey topic information.

Next, in step 940, selected image units, e.g., the image units not discriminated in step 930, are evaluated, without decoding the image units being classified or reference to decoded image data, based on an evaluation of predetermined morphological (structural) image characteristics of the image units. The evaluation entails a determination (step 941) of the morphological image characteristics and a comparison (step 942) of the determined morphological image characteristics for each image unit. The determined morphological image characteristic(s), e.g., the derived image unit shape representations, of each selected image unit are compared, either with the determined morphological image characteristic(s)/derived image unit shape representations of the other selected image units (step 942A), or with predetermined/user-selected morphological image characteristics to locate specific types of image units (step 942B). The determined morphological image characteristics of the selected image units are advantageously compared with each other for the purpose of identifying equivalence classes of image units such that each equivalence class contains most or all of the instances of a given image unit in the document, and the relative frequencies with which image units occur in a document can be determined.

It will be appreciated that the specification of the morphological image characteristics for titles, headings, captions, linguistic criteria or other significance indicating features of a document image can be predetermined and selected by the user to determine the selection criteria defining a "significant" image unit. Comparing the image characteristics of the selected image units of the document image for matches with the image characteristics associated with the selection criteria permits the significant image units to be readily identified without any document decoding.

Any of a number of different methods of comparison can be used. One technique that can be used, for example, is by correlating the raster images of the extracted image units using decision networks, such technique being described for characters in a Research Report entitled "Unsupervised Construction of Decision Networks for Pattern Classification" by Casey et al., IBM Research Report, 1984, incorporated herein in its entirety.

Another techniques that can be used to identify equivalence classes of word units are the word shape comparison techniques disclosed in U.S. patent application Ser. Nos. 07/796,119 and 07/795,169, filed concurrently herewith by Huttenlocher and Hopcroft, and by Huttenlocher, Hopcroft and Wayner, respectively, and entitled, respectively, "Optical Word Recognition By Examination of Word Shape," and "Method for Comparing Word Shapes." This method provides an adequate comparison for purposes of determining phrase frequency is to compare only the length and height of the derived image unit shape representations. Such a comparison is particularly fast, resulting in a highly efficient phrase frequency analysis which has proven to be sufficiently robust to reliably extract significant phrases in many text document applications.

In instances in which multiple page documents are processed, each page is processed and the data held in the memory 15 (see FIG. 1), as described above. The entirety of the data can then be processed.

The second phase of the document analysis according to this method involves further processing (step 950) of the scanned document image to emphasize the identified image units. The emphasis can be provided in numerous ways. One exemplary way is to augment the document image so that the identified significant image units are underscored, highlighted with color, or presented as margin notations.

Another exemplary way is to modify the shape and/or other appearance attributes of the significant image units themselves in a manner which emphasizes them relative to the other image units in the document image. The appearance modification can be accomplished using any conventional image modification techniques, or, advantageously, the following morphological bitmap modification techniques.

In accordance with this method, one or more selected morphological operations are performed uniformly on the entire bitmap for a selected image unit to modify at least one shape characteristic thereof. It will be appreciated that the selection of bitmap operations may be performed automatically or interactively.

Examples of ways in which the appearance changes described above can be accomplished are as follows. The type style text can be "boldened" by either "dilation" or using a connectivity-preserving (CP) thickening operation. It can be "lightened" by either "erosion" or a CP thinning operation. (As will be appreciated by those skilled in the art, dilation and erosion are morphological operations which map a source image onto an equally sized destination image according to a rule defined by a pixel pattern called a structuring element (SE). A SE is defined by a center location and a number of pixel locations, each having a defined value (ON or OFF). The pixels defining the SE do not have to be adjacent each other. The center location need not be at the geometrical center of the pattern; indeed it need not even be inside the pattern. In a dilation, a given pixel in the source image being ON causes the SE to be written into the destination image with the SE center at the corresponding location in the destination image. The SEs used for dilation typically have no OFF pixels. In an erosion, a given pixel in the destination image is turned ON if and only if the result of superimposing the SE center on the corresponding pixel location in the source image results in a match between all ON and OFF pixels in the SE and the underlying pixels in the source image.)

Such dilation/thickening and erosion/thinning operations can be either isotropic (the same horizontally for vertically) or anisotropic (e.g., different in horizontal and vertical directions).

Although optical character recognition (OCR) techniques are required, for example, in order to convert the typestyle

of a selected word unit to italic, a similar type of emphasis can be achieved through the morphological operation of horizontal shearing to achieve the slant typestyle. Slant is a variant of roman type style that is created from roman using a horizontal shear of about 12 degrees (this is the approximate slant angle of italic style characters). The sheared images can slant forwards, backwards, or even upwards, if desired. Text can also be bit inverted (black for white and vice versa) for emphasis, or words can be emphasized or de-emphasized by scaling up or down, respectively. In the case of scaling, it may also be desirable to change the thickness of the lines in the image unit in addition to simple scaling.

Thus, using such morphological bitmap alteration processes, hand marks such as underlining, side lining, circling, highlighting, and so forth, can be extracted from the image, and removed from the original bitmap by XOR operations. Removal of color highlight marks requires capture of a gray scale (or color) scanned image. Once captured, removal is relatively easy using the appropriate thresholding. The resulting image is similar in quality to that of un-highlighted marks. Words that are high-lighted can be identified from the highlight mask and word boxes, using known seed-growing methods. The appearance of these words can be altered at will.

A salient feature provided by the method of the invention is that the initial processing and identification of significant image units is accomplished without an accompanying requirement that the content of the image units be decoded, or that the information content of the document image otherwise be understood. More particularly, to this stage in the process, the actual content of the word units is not required to be specifically determined. Thus, for example, in such applications as copier machines or electronic printers that can print or reproduce images directly from one document to another without regard to ASCII or other encoding/decoding requirements, image units can be identified and processed using one or more morphological image characteristics or properties of the image units. The image units of unknown content can then be further optically or electronically processed. One of the advantages that results from the ability to perform such image unit processing without having to decode the image unit contents at this stage of the process is that the overall speed of image handling and manipulation can be significantly increased.

The second phase of the document analysis of the invention involves processing (step 50) the identified significant image units to produce an auxiliary or supplemental document image reflective of the contents of the source document image. It will be appreciated that the format in which the identified significant image units are presented can be varied as desired. Thus, the identified significant image units could be presented in reading order to form one or more phrases, or presented in a listing in order of relative frequency of occurrence. Likewise, the supplemental document image need not be limited to just the identified significant image units. If desired, the identified significant image units can be presented in the form of phrases including adjacent image units presented in reading order sequence, as determined from the document location information derived during the document segmentation and structure determination steps 20 and 25 described above. Alternatively, a phrase frequency analysis as described above can be conducted to limit the presented phrases to only the most frequently occurring phrases.

The present invention is similarly not limited with respect to the form of the supplemental document image. One application for which the information retrieval technique of the invention is particularly suited is for use in reading machines for the blind. One embodiment supports the designation by a user of key words, for example, on a key word list, to designate likely points of interest in a document. Using the user designated key words, occurrences of the word can be found in the document of interest, and regions of text forward and behind the key word can be retrieved and processed using the techniques described above. Or, as mentioned above, significant key words can be automatically selected according to prescribed criteria, such as frequency of occurrence, or other similar criteria, using the morphological image recognition techniques described above; and a document automatically summarized using the determined words.

Another embodiment supports an automatic location of significant segments of a document according to other predefined criteria, for example, document segments that are likely to have high informational value such as titles, regions containing special font information such as italics and boldface, or phrases that receive linguistic emphasis. The location of significant words or segments of a document may be accomplished using the morphological image recognition techniques described above. The words thus identified as significant words or word units can then be decoded using optical character recognition techniques, for example, for communication to the blind user in a Braille or other form which the blind user can comprehend. For example, the words which have been identified or selected by the techniques described above can either be printed in Braille form using an appropriate Braille format printer, such as a printer using plastic-based ink; or communicated orally to the user using a speech synthesizer output device.

Once a condensed document is communicated, the user may wish to return to the original source to have printed or hear a full text rendition. This may be achieved in a number of ways. One method is for the associated synthesizer or Braille printer to provide source information, for example, "on top of page 2 is an article entitled . . . " The user would then return to point of interest.

Two classes of apparatus extend this capability through providing the possibility of user interaction while the condensed document is being communicated. One type of apparatus is a simple index marker. This can be, for instance, a hand held device with a button that the user depresses whenever he or she hears a title of interest, or, for instance, an N-way motion detector in a mouse **19** (FIG. 2) for registering a greater variety of commands. The reading machine records such marks of interest and returns to the original article after a complete summarization is communicated.

Another type of apparatus makes use of the technology of touch-sensitive screens. Such an apparatus operates by requiring the user to lay down a Braille summarization sheet **41** on a horizontal display. The user then touches the region of interest on the screen **42** in order to trigger either a full printout or synthesized reading. The user would then indicate to the monitor when a new page was to be processed.

It will be appreciated that the method of the invention as applied to a reading machine for the blind reduces the amount of material presented to the user for evaluation, and thus is capable of circumventing many problems inherent in the use of current reading technology for the blind and others, such as the problems associated with efficient browsing of a document corpus, using synthesized speech, and the problems created by the bulk and expense of producing Braille paper translations, and the time and effort required by the user to read such copies.

The present invention is useful for forming abbreviated document images for browsing (image gists). A reduced representation of a document is created using a bitmap image of important terms in the document. This enables a user to quickly browse through a scanned document library, either electronically, or manually if summary cards are printed out on a medium such as paper. The invention can also be useful for document categorization (lexical gists). In this instance, key terms can be automatically associated with a document. The user may then browse through the key terms, or the terms may be further processed, such as by decoding using optical character recognition.

Although the invention has been described and illustrated with a certain degree of particularity, it is understood that the present disclosure has been made only by way of example, and that numerous changes in the combination and arrangement of parts can be resorted to by those skilled in the art without departing from the spirit and scope of the invention, as hereinafter claimed.

APPENDIX

Jul 26 19:28 1991 args.h

```
/* Support for command line argument scanning.
 *
 * When a program is run from the shell, its name is followed by a number of
 * required command line ARGUMENTS and then some optional command line OPTIONS.
 * Each argument consists of a list of required PARAMETERS, each of which can
 * be either an int, string, or float. Options are like arguments with the
 * exception that their required parameters are predeced by a keyword denoting
 * which option is being invoked.
 *
 * Required arguments are defined using the DefArg function. The format string
 * consists of a list of data format specifiers (%d, %f, and %s for integer, float, and
 * string, respectively) that specify the types of the parameters to the arguement.
 * The documentation string should contain a one line description of the argument.
 * It will be printed if the argument list cannot be scanned.
 * The remaining arguments to DefArg are pointers to locations where the values of the
 * command line arguments will be stored.
 *
 * Optional arguments are defined with the DefOption function. The format string
 * is similar to the DefArg format string, but has a keyword before the format
 * specifiers. The exists parameter is a pointer to BOOLEAN that is set to true
 * iff an occurance of this option was successfully parsed from the command line.
 * The remaining arguments are pointers to the locations where the values of the
 * command line arguments will be stored.
 *
 * Short example:
 * The following program expects one required command line argument that is a string
 * and will be stored in s. In addition, it will accept three different optional
 * keyword arguments. They are the keyword -int followed by an integer, with result
 * stored in i; -float followed by a float stored in f; and -pair followed by a float
 * and then an int, stored in f and i, respectively.
 *
 * Suppose the program is called foo. Here are some legal invocations:
 * % foo hello
 * % foo hello -int 1
 * % foo hello -int 5 -float 10
 * % foo hello -pair 1 2
 *
 * Here are some error invocations and responses
 * % foo
 * Usage:
 *  scanArgs
 *    filename
 *    [-int <int>]
 *    [-float <float>]
 *    [-pair <float> <int>]
 * % foo hello -int
 * Option -int expects 1 parameters:
 * -int <int>
 *
 *
 *void main (int argc,char **argv)
```

```
*{
* int i;
* float f;
* char *s;
* BOOLEAN haveAString,haveAnInt,haveAFloat,haveAPair;
*
* DefArg("%s","filename",&s);
* DefOption("-int %d","-int <int>",&haveAnInt,&i);
* DefOption("-float %f","-float <float>",&haveAFloat,&f);
* DefOption("-pair %f %d","-pair <float> <int>",&haveAPair,&f,&i);
*
* ScanArgs(argc,argv);
*
* printf("%s\n",s);
* if (haveAPair)
*   printf("%f %d\n",f,i);
* if (haveAnInt)
*   printf("%d\n",i);
* if (haveAFloat)
*   printf("%f\n",f);
* if (haveAString)
*   printf("%s\n",s);
*}
*
*/

/* Possible additions:
 *  1) Passing NULL pointers as exists variables.
 *  2) Predicate calculus for error checking.
 *  3) Only need one DefArg call.
 *  4) Combine with error.c to save program name info.
 */
void DefArg(char *format,char *documentation,...);
void DefOption(char *format,char *documentation,BOOLEAN *exists,...);
void ScanArgs(int argc,char **argv);
```

Section A APPENDIX / Page 3

Jan 11 17:00 1991 baselines.h

```
List BaseLines(Picture pict,double angle,char *plotFile);
#ifdef foo
int *count,
        int **returnCoordx, int **returnCoordy);
#endif
void DrawBaseLines(Picture pict, List pointList, double angle);
```

Section A APPENDIX / Page 4

Aug 23 13:03 1991 blobify.h

```
Picture Blobify(Picture old,int half_mask_size,double threshold);
Picture NewBlobify(Picture old,int halfMaskWidth,double threshold,double angle);
```

Aug 1 02:59 1991 boolean.h

```
typedef int BOOLEAN;
#define FALSE 0
#define TRUE (!FALSE)
```

Section A APPENDIX / Page 6

Jan 11 17:00 1991 boxes.h

```
List FindBorders(Picture pict,double theta);
void DrawBox(Picture pict,Box box);
void DrawColorBox(Picture pict,Box box,int color);
```

Jul 26 13:42 1991 descriptors.h

```
typedef unsigned char *Descriptor,DescriptorElement;

void PrintField(char *s,int w);
void PrintDescriptor(Descriptor d,int *starCount,int *correctCount);
void PrintWords(char **words,int numberOfWords);
Descriptor ComputeDescriptor(int modelIndex,Dictionary models,
                        Dictionary thisFont,int numberOfWords,
                        DiffDescriptor dd);

#define MAX_FONTS (20)
#define MAX_WORDS (100)
```

Jan 16 12:55 1991 dict.h

```
/* Dictionary file have the following format:
 * int magic number = 1234567
 * int numberOfEntries
 * int infoStringLength (includes the \0 at the end)
 * char infoString[infoStringLength]
 * OutlinePairBody[numberOfEntries]
 */

typedef struct {
  Box box;
  float blackoutHeight;
  int numberOfLegs;
  int offset;
  int width;
  float *x;
  float *top;
  float *bottom;
} *OutlinePair,OutlinePairBody;

typedef struct {
  Box box;
  int numberOfLegs;
  int *x;
  int *top;
  int *bottom;
} *RawOutlinePair,RawOutlinePairBody;

typedef struct {
  int numberOfEntries;
  char *infoString;
  RawOutlinePair *rawOutlines;
  OutlinePair *outlines;
} *Dictionary,DictionaryBody;

void WriteDictionary(Dictionary dict, char *filename);
Dictionary ReadDictionary(char *filename);
Dictionary NewDict(int numberOfEntries);
char *ArgListToString(int argc, char **argv);
```

Section A APPENDIX / Page 9

Jul 30 23:04 1991 diff.h

```
typedef enum {L2,CONSTRAINED,WARP} DiffType;

typedef struct {
 DiffType diffType;
 BOOLEAN lengthNormalize;
 BOOLEAN separate;
 float centerWeight;
 int bandWidth;
 float topToBottom;
 float hillToValley;
 FILE *pathFP;
} *DiffDescriptor,DiffDescriptorBody;

Picture CompareDictionaries(Dictionary dict1, Dictionary dict2,DiffDescriptor dd);
void WritePictureAsAscii(Picture pict, char *filename,
                    char *info1, char *info2);
float DiffPair(OutlinePair one, OutlinePair two,DiffDescriptor dd);
#ifdef foo
float DiffPairAndPath(OutlinePair one, OutlinePair two,DiffDescriptor dd);
#endif
```

Section A

APPENDIX / Page 10

Jan 15 18:56 1991 diff2.h

```
#ifdef OWNER
#define EXTERN
#else
#define EXTERN extern
#endif OWNER

EXTERN int FileCountX;
EXTERN int FileCountY;

float DiffPair(OutlinePair one, OutlinePair two, char *matchtype,
        char *pathFile);

```

Jul 26 19:29 1991 error.h

```
/* Possible additions:
 * 1) Variable numbers of parameters to DoError().
 * 2) Error recovery language.
 */
void DoError(char *string1,char *string2);
```

Aug 15 06:37 1991 fontNorm.h

```
void StoreRawOutlinePair(Dictionary dict, int dictEntry,
                Box box,int *bothX,int *topY, int *baseY,
                int numberOfLegs);

#define HIT_THE_BOX (10000)

```

Jan 11 17:00 1991 lines.h

```
typedef BOOLEAN pistFunc(Picture pict, int x, int y, BOOLEAN test,
                        UCHAR color);

pistFunc DrawPiston, CountPiston, DistancePiston, BaseLinePiston;

void LineEngine(Picture pict, int x1, int y1, int x2, int y2, UCHAR color,
                pistFunc PerPixel);
void DrawLine(Picture pict, int x1, int y1, int x2, int y2, UCHAR color);
float CountLine(Picture pict, int x1, int y1, int x2, int y2);
int DistanceLine(Picture pict, int x1, int y1, int x2, int y2);
```

Section A APPENDIX / Page 14

Jan 11 17:00 1991 lists.h

```
typedef struct {
  void *car;
  void *cdr;
} cellBody,*cell;

typedef cell List;
typedef void *mapFun(void *);
typedef void collectFun(void *);

List cdr(List);
void *car(List);
void *popIntern(List *);
BOOLEAN endp(List);
List cons(void *,List);
void map(List,mapFun);
List collect(List,collectFun);
int ListLength(List l);

#define push(a,l) ((l) = cons((a),(l)))
#define pop(l) (popIntern(&(l)))
#define nil ((List)NULL)
```

Jan 15 18:39 1991 match.h

```
#ifdef OWNER
#define EXTERN
#else
#define EXTERN extern
#endif OWNER

EXTERN int debug;

typedef struct {
  float cost;
  int xptr;
  int yptr;
} elt;

#define MAXSEQLENGTH 800

float DPDiffPair(OutlinePair one, OutlinePair two);
float matchvecs(float *Vec1, int lenVec1, float *Vec2, int lenVec2);
float sq_distance(float x1, float x2);
float best_score (elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2);
void print_best_path(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2,
                char *pathFile);
void print_array_costs(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2);
void print_array_dirs(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2);

/*
#ifndef debug
#define debug FALSE
#endif
*/
#ifndef horweight
#define horweight 1.5
#endif

#ifndef verweight
#define verweight 1.5
#endif

#ifndef diagweight
#define diagweight 1.0
#endif
```

Jan 15 18:47 1991 matchparallel.h

```
float pl_DPDiffPair(OutlinePair one, OutlinePair two, char *pathFile);
float pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1, float *Vec2t, float *Vec2b, int lenVec2, char *pathFile);

float faster_pl_DPDiffPair(OutlinePair one, OutlinePair two, char *pathFile);
float faster_pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1, float *Vec2t, float *Vec2b, int lenVec2, char *pathFile);

float simple_pl_DPDiffPair(OutlinePair one, OutlinePair two);
float simple_pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1, float *Vec2t, float *Vec2b, int lenVec2);
```

Jul 9 16:01 1991 misc.h

```
/*
 *
 * misc.h - miscellaneous types and declarations
 *
 */

/* Some library routines that never seem to get declared */

/* Memory allocation functions */
extern void *malloc(unsigned size);
extern void *calloc(unsigned nelem, unsigned elsize);
extern void *realloc(void *p, unsigned size);
extern void free(void *p);

/* I don't feel like including setjmp.h */
/*
extern int _setjmp(jmp_buf env);
extern volatile void _longjmp(jmp_buf env, int val);
*/

/* String-to-X functions */
extern int atoi(char *s);
extern double atof(char *s);

/* String functions */
extern int strcmp(char *s1, char *s2);
extern int strncmp(char *s1, char *s2, int n);
extern char *strcpy(char *d, char *s);
extern char *strncpy(char *d, char *s, int n);
int strlen(char *s);
extern char *strdup(char *);
extern char *strchr(char *s,char c);

/* stdio functions */
extern int fclose(FILE *stream);
extern int fread(char *ptr, int size, int nitems, FILE *stream);
extern int fwrite(char *ptr, int size, int nitems, FILE *stream);
/* these are necessary to avoid implicit declarations */
extern int _flsbuf();
extern int _filbuf();

/* Formatted I/O functions */
extern int printf(char *format, ...);
extern int scanf(char *format, ...);
extern int fprintf(FILE *stream, char *format, ...);
extern int fscanf(FILE *stream, char *format, ...);

/* and misc stuff */
extern volatile void exit(int val);

extern void perror(char *s);
```

Section A APPENDIX / Page 18

Aug 1 02:59 1991 mylib.h

```
#include "error.h"
#include "boolean.h"
#include "lists.h"
#include "args.h"
#include "pict.h"
#include "read.h"
```

Aug 15 06:36 1991 newContour.h

```
void BoxToShell(Picture pict,Box box,List baseLinePoints,
                Dictionary dict,int dictEntry,NormalizationDescriptor *nd);
void BarBoxList(Picture pict,List boxList,List baseLinePoints,
                char *filename, char *infoString, NormalizationDescriptor *nd);
```

Jul 31 17:11 1991 newMatch.h

```
extern float hillToValley;
extern float L2Compare(OutlinePair o1,OutlinePair o2,float topToBottom);
extern float NewMatch(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
   float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth,
   float topToBottom);
extern float SepMatch(float *a1,int aLength,float *b1,int bLength,
   float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth);
extern float NewMatchAndPath(float *a1,float *a2,int aLength,float *b1,float *b2,
   int bLength,float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth,
   float topToBottom,FILE *fp);
extern float SlopeCMatch(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
  float centerWeight,BOOLEAN lengthNormalize,float topToBottom);
extern float SepSlopeCMatch(float *a1,int aLength,float *b1,int bLength,
   float centerWeight,BOOLEAN lengthNormalize);
extern float SlopeCMatchAndPath(float *a1,float *a2,int aLength,float *b1,float *b2,
   int bLength,float centerWeight,BOOLEAN lengthNormalize,float topToBottom,
   FILE *pathFP);
```

Jan 11 17:00 1991 numbers.h

```
void DrawNumber(Picture pict, int x, int y, int color, float scale, int n);
```

Section A APPENDIX / Page 22

Jan 14 16:52 1991 orient.h

```
BOOLEAN Coarse(Picture pict, int coarseSamples, int coarseDirections,
        float *orientation, char *plotFile);

float Fine(Picture pict,int fineSamples, int fineDirections,
        int coarseDirections, float coarseAngle, char *plotFile);

float NewFine(Picture pict,int fineSamples, int fineDirections,
        float angleStart,float angleEnd, char *plotFile);
```

Aug 23 19:19 1991 pict.h

```
typedef unsigned char UCHAR;

#define ROUND8(x) ((x%8)?(x+8-x%8):x)
#define ROUND16(x) ((x%16)?(x+16-x%16):x)
#define ROUND2(x) ((x%2)?(x+1):x)

typedef int Color;
#define COLOR_RED 0
#define COLOR_GREEN 1
#define COLOR_BLUE 2

typedef struct cmapstruct {
 int numberOfEntries;
 UCHAR *red;
 UCHAR *green;
 UCHAR *blue;
} ColorMapBody, *ColorMap;

typedef struct pstruct {
 int width;
 int height;
 int depth;
 int uchar_width;
 ColorMap cmap;
 UCHAR *data;
} PictureBody, *Picture;

void doerror(char *string1,char *string2);


ColorMap NewColorMap(int size);
void FreeColorMap(ColorMap cmap);
UCHAR ReadColorValue(ColorMap cmap, Color primary,int index);
UCHAR WriteColorValue(ColorMap cmap, int index, UCHAR red, UCHAR green,
                  UCHAR blue);
Picture new_pict(int width,int height,int depth);
void free_pict(Picture pict);
Picture load_pict(char *filename);
Picture load_header(FILE *fp);
void write_pict(char *filename,Picture pict);
void write_header(FILE *fp, Picture pict);
/*int BytesPerScanline(Picture pict); */
#define BytesPerScanline(pict) (pict->uchar_width)

UCHAR ReadPixel(Picture pict,int x,int y);
void WritePixel(Picture pict,int x,int y,int color);
void WriteClippedPixel(Picture pict,int x,int y,int color);
void CopyPicture(Picture dest, Picture src);
```

Section A APPENDIX / Page 24

Jul 26 13:09 1991 read.h

```
int ReadInt(FILE *fp);
int ReadFloat(FILE *fp);
char *ReadString(FILE *fp);
```

Aug 15 00:19 1991 types.h

```
typedef struct {
 BOOLEAN noAscenderNormalize;
 BOOLEAN noXHeightNormalize;
} NormalizationDescriptor;

typedef struct {
 int x;
 int y;
 int width;
 int height;
 int pageX;
 int pageY;
 double angle;
} BoxBody, *Box;

typedef struct {
 int x;
 int y;
} PointBody,*Point;

Box MakeBox(int x,int y,int width,int height,double angle);
Point MakePoint(int x,int y);
```

Jul 26 13:25 1991 Makefile

```
CCFLAGS = -g -c -I/net/piglet/piglet-1c/hopcroft/new/include

INCLUDE = /net/piglet/piglet-1c/hopcroft/new/include/

ARGS = $(INCLUDE)args.h
BOOLEAN = $(INCLUDE)boolean.h
ERROR = $(INCLUDE)error.h
LISTS = $(INCLUDE)lists.h
MISC = $(INCLUDE)misc.h
PICT = $(INCLUDE)pict.h
READ = $(INCLUDE)read.h

OFUNS = args.o error.o pict.o lists.o read.o

mylib.a: $(OFUNS)
	ld -r $(OFUNS) -o mylib.a

args.o:		args.c $(BOOLEAN) $(ERROR) $(MISC) $(ARGS)
	gcc $(CCFLAGS) args.c

error.o:	error.c $(ERROR)
	gcc $(CCFLAGS) error.c

pict.o:			pict.c $(BOOLEAN) $(ERROR) $(PICT)
	gcc $(CCFLAGS) pict.c

lists.o:	lists.c $(BOOLEAN) $(LISTS)
	gcc $(CCFLAGS) lists.c

read.o:			read.c $(MISC) $(READ)
	gcc $(CCFLAGS) read.c
```

Jul 26 13:23 1991 args.c

```
#include <stdio.h>
#include <stdarg.h>
#include "error.h"
#include "boolean.h"
#include "misc.h"
#include "args.h"

#define MAX_NAME_LENGTH (50)
#define MAX_PARAMETERS (6)
#define MAX_OPTIONS (20)
#define MAX_ARGS (20)

typedef enum {INTEGER,FLOAT,STRING} ParamType;

typedef struct {
 char *documentation;
 int numberOfParameters;
 ParamType types[MAX_PARAMETERS];
 void *values[MAX_PARAMETERS];
} *Arg,ArgBody;

typedef struct {
 char optionName[MAX_NAME_LENGTH+1];
 char *documentation;
 BOOLEAN *exists;
 int numberOfParameters;
 ParamType types[MAX_PARAMETERS];
 void *values[MAX_PARAMETERS];
} *Option,OptionBody;

static BOOLEAN optionsRequired = TRUE;
static int numberOfArguments = 0;
static ArgBody args[MAX_ARGS];
static int numberOfOptions = 0;
static OptionBody options[MAX_OPTIONS];

void DefArg(char *format,char *documentation,...)
{
 va_list ap;
 char *p;
 int i;
 int parameterCounter;

 if (numberOfArguments==MAX_ARGS)
  DoError("Def Arg: too many command line options now:\"%s\".\n",format);

 args[numberOfArguments].documentation = documentation;

 /* now parse the format string */
 /* get option parameters */
 va_start(ap,documentation);
 for (p=format,parameterCounter=0;*p;p++) {
```

```
    if (*p=='%'){
     if (parameterCounter == MAX_PARAMETERS)
        DoError("Def Arg: too many parameters in \"%s\".\n",format);
     p++;
     switch (*p) {
     case 'd':
        args[numberOfArguments].types[parameterCounter] = INTEGER;
      args[numberOfArguments].values[parameterCounter] = va_arg(ap, void *);
        parameterCounter++;
        break;
     case 'f':
        args[numberOfArguments].types[parameterCounter] = FLOAT;
      args[numberOfArguments].values[parameterCounter] = va_arg(ap, void *);
        parameterCounter++;
        break;
     case 's':
        args[numberOfArguments].types[parameterCounter] = STRING;
      args[numberOfArguments].values[parameterCounter] = va_arg(ap, void *);
        parameterCounter++;
        break;
     default:
        DoError("DefArg: bad option in \"%s\".\n",format);
     }
    }
   }
   args[numberOfArguments].numberOfParameters = parameterCounter;
   ++numberOfArguments;
   va_end(ap);
  }

  void DefOption(char *format,char *documentation,BOOLEAN *exists,...)
  {
   va_list ap;
   char *optionName;
   char *p;
   int i;
   int parameterCounter;

   if (numberOfOptions==MAX_OPTIONS)
    DoError("DefOption: too many command line options now:\"%s\".\n",format);

   /* record exists so that *exists will be TRUE if this option is scanned */
   options[numberOfOptions].exists = exists;

   options[numberOfOptions].documentation = documentation;

   /* now parse the format string */
   p=format;
   /* skip leading spaces */
   while (*p==' ' && *p!='\0')
    p++;

   /* get the option name */
   optionName = options[numberOfOptions].optionName;
   i=0;
```

```
 while (*p != '\0' && *p != ' ' && *p !='\t') {
  if (i<MAX_NAME_LENGTH)
   optionName[i++] = *p;
  else
   DoError("DefOptions: option name too long in \"%s\".\n",format);
  p++;
 }
 optionName[i] = '\0';

 /* get option parameters */
 va_start(ap,exists);
 for (parameterCounter=0;*p;p++) {
  if (*p=='%') {
   if (parameterCounter == MAX_PARAMETERS)
       DoError("DefOptions: too many parameters in \"%s\".\n",format);
   p++;
   switch (*p) {
   case 'd':
       options[numberOfOptions].types[parameterCounter] = INTEGER;
    options[numberOfOptions].values[parameterCounter] = va_arg(ap, void *);
       parameterCounter++;
       break;
   case 'f':
       options[numberOfOptions].types[parameterCounter] = FLOAT;
    options[numberOfOptions].values[parameterCounter] = va_arg(ap, void *);
       parameterCounter++;
       break;
   case 's':
       options[numberOfOptions].types[parameterCounter] = STRING;
    options[numberOfOptions].values[parameterCounter] = va_arg(ap, void *);
       parameterCounter++;
       break;
   default:
       DoError("DefOptions: bad option in \"%s\".\n",format);
   }
  }
 }
 options[numberOfOptions].numberOfParameters = parameterCounter;
 ++numberOfOptions;
 va_end(ap);
}

void PrintHelp(char *name)
{
 int i;
 fprintf(stderr,"Usage:\n %s\n",name);
 for (i=0;i<numberOfArguments;++i)
  fprintf(stderr,"   %s\n",args[i].documentation);
 for (i=0;i<numberOfOptions;++i)
  fprintf(stderr,"   [%s]\n",options[i].documentation);
 DoError("\n",NULL);
}

void ScanArgs(int argc,char **argv)
{
```

```
int i,j,k;

for (j=0;j<numberOfOptions; ++j)
 *(options[j].exists) = FALSE;

if (argc == 1 && optionsRequired)
 PrintHelp(argv[0]);

i=1;
for (j=0;j<numberOfArguments; ++j) {
 if (i+args[j].numberOfParameters > argc) {
  fprintf(stderr,"Required argument expects %d parameters:\n %s\n",
         args[j].numberOfParameters,
         args[j].documentation);
  DoError("\n",NULL);
 }
 for (k=0;k<args[j].numberOfParameters; ++k)
  switch (args[j].types[k]) {
  case INTEGER:
      *(int *)(args[j].values[k]) = atoi(argv[i++]);
      break;
  case FLOAT:
      *(float *)(args[j].values[k]) = atof(argv[i++]);
      break;
  case STRING:
      *(char **)(args[j].values[k]) = argv[i++];
      break;
  default:
      DoError("ScanArgs: internal error - bad type.\n",NULL);
  }
}

while (i<argc) {
 for (j=0;j<numberOfOptions; ++j)
  if (!strcmp(options[j].optionName,argv[i])) {
      if (i+options[j].numberOfParameters >= argc) {
       fprintf(stderr,"Option %s expects %d parameters:\n %s\n",
              options[j].optionName,
              options[j].numberOfParameters,
              options[j].documentation);
       DoError("\n",NULL);
      }
      *(options[j].exists) = TRUE;
      ++i;
      for (k=0; k<options[j].numberOfParameters; ++k)
       switch (options[j].types[k]) {
       case INTEGER:
        *(int *)(options[j].values[k]) = atoi(argv[i++]);
        break;
       case FLOAT:
        *(float *)(options[j].values[k]) = atof(argv[i++]);
        break;
       case STRING:
        *(char **)(options[j].values[k]) = argv[i++];
        break;
```

```
        default:
         DoError("ScanArgs: internal error - bad type.\n",NULL);
        }
       break;
    }
   if (j= =numberOfOptions) {
    fprintf(stderr,"Bad command line argument.\n");
    PrintHelp(argv[0]);
   }
  }
}
#ifdef foo
void main (int argc,char **argv)
{
 int i;
 float f;
 char *s;
 BOOLEAN haveAString,haveAnInt,haveAFloat,haveAPair;

 DefArg("%s","filename",&s);
 DefOption("-int %d","-int <int>",&haveAnInt,&i);
 DefOption("-float %f","-float <float>",&haveAFloat,&f);
 DefOption("-pair %f %d","-pair <float> <int>",&haveAPair,&f,&i);

 ScanArgs(argc,argv);

 printf("%s\n",s);
 if (haveAPair)
  printf("%f %d\n",f,i);
 if (haveAnInt)
  printf("%d\n",i);
 if (haveAFloat)
  printf("%f\n",f);
 if (haveAString)
  printf("%s\n",s);
}
#endif
```

Jul 26 12:57 1991 error.c

```
#include <stdio.h>
#include "error.h"

void DoError(char *string1,char *string2)
{
 if (string2 == NULL)
  printf(string1);
 else
  printf(string1,string2);
 exit(-1);
}

```

Jul 26 12:57 1991 lists.c

```
#include "stdio.h"
#include "boolean.h"
#include "lists.h"

List cdr(List l)
{
 if (l == NULL)
  return l;
 else
  return l->cdr;
}

void *car(List l)
{
 if (l == NULL)
  return l;
 else
  return l->car;
}

void *popIntern(List *l)
{
 List temp;
 if (*l == NULL)
  return *l;
 else {
  temp = (*l)->car;
  *l = (*l)->cdr;
  return temp;
 }
}

BOOLEAN endp(List l)
{
 return (l == NULL);
}

List cons(void *theCar,List theCdr)
{
 cell temp;
 temp = (cell)calloc(1,sizeof(cellBody));
 if (temp == NULL) {
  printf("Cons: out of memory\n");
  exit(-1);
 }
 temp->car = theCar;
 temp->cdr = theCdr;
 return temp;
}

void map(List l,mapFun f)
{
```

```
  while (l != NULL) {
   (*f)(l->car);
   l = l->cdr;
  }
}

List collect(List l,collectFun c)
{
  List temp;
  while (l!=NULL) {
   (*c)(l->car);
   temp = l;
   l = l->cdr;
   free(temp);
  }
}

int ListLength(List l)
{
  int count=0;
  while (l != NULL) {
   ++count;
   l = l->cdr;
  }
  return count;
}
```

Aug 23 19:20 1991 pict.c

```
#include <stdio.h>
#include <math.h>
#include <rasterfile.h>
#include "boolean.h"
#include "error.h"
#include "pict.h"

static UCHAR bitmasks[] = { 0x80,0x40,0x20,0x10,0x8,0x4,0x2,0x1 };

ColorMap NewColorMap(int size)
{
 ColorMap cmap;
 if (size > 256)
  DoError("NewColorMap: size greater than 256.",NULL);
 if (size < 1)
  DoError("NewColorMap: size less than 1.",NULL);
 if ((cmap = (ColorMap)calloc(1,sizeof(ColorMapBody))) == NULL)
  DoError("NewColorMap: cannot allocate space.",NULL);
 cmap->numberOfEntries = size;
 cmap->red = (UCHAR *)calloc(size,sizeof(UCHAR));
 cmap->green = (UCHAR *)calloc(size,sizeof(UCHAR));
 cmap->blue = (UCHAR *)calloc(size,sizeof(UCHAR));
 if ((cmap->red == NULL)||(cmap->green == NULL)||(cmap->blue == NULL))
  DoError("NewColorMap: cannot allocate space.",NULL);
 return cmap;
}

void FreeColorMap(ColorMap cmap)
{
 if (cmap != NULL) {
  if (cmap->red != NULL)
   free(cmap->red);
  if (cmap->green != NULL)
   free(cmap->green);
  if (cmap->blue != NULL)
   free(cmap->blue);
  free(cmap);
 }
}

UCHAR ReadColorValue(ColorMap cmap, Color primary, int index)
{
 if (index > cmap->numberOfEntries)
  DoError("ReadColorValue: index too big.",NULL);
 if (primary = COLOR_RED)
  return *(cmap->red+index);
 if (primary = COLOR_GREEN)
  return *(cmap->green+index);
 if (primary = COLOR_BLUE)
  return *(cmap->blue+index);
 DoError("ReadColorValue: bad primary color.",NULL);
}
```

```
UCHAR WriteColorValue(ColorMap cmap, int index, UCHAR red, UCHAR green,
                 UCHAR blue)
{
 if (index > cmap->numberOfEntries)
  DoError("WriteColorValue: index too big.",NULL);
 *(cmap->red+index)=red;
 *(cmap->green+index)=green;
 *(cmap->blue+index)=blue;
}

Picture new_pict(width,height,depth)
int width,height,depth;
{
 Picture pict;
 int uchar_width;

 if ((pict = (Picture)calloc(1,sizeof(PictureBody))) == NULL)
  DoError("new_pict: cannot allocate space",NULL);
 pict->width = width;
 pict->height = height;
 pict->depth = depth;
 pict->cmap = NULL;
 if (pict->depth == 32)
  uchar_width = pict->width*4;
 else if (pict->depth == 8)
  uchar_width = ROUND2(pict->width);
 else if (pict->depth == 1)
  uchar_width = ROUND16(pict->width) >> 3;
 else
  DoError("new_pict: only depths of 1 and 8 are supported\n",NULL);
 pict->uchar_width = uchar_width;

 pict->data = (UCHAR *) calloc(uchar_width * pict->height , sizeof(UCHAR));
 if (pict->data == NULL)
  DoError("new_pict: cannot allocate space\n",NULL);
 return pict;
}

void free_pict(pict)
Picture pict;
{
 if (pict->data != NULL)
  free(pict->data);
 FreeColorMap(pict->cmap);
 free(pict);
}

Picture load_pict(fn)
char *fn;
{
 FILE *fp;
 Picture pict;
 int uchar_width;
 struct rasterfile header;
```

```

 if ((pict = (Picture)calloc(1,sizeof(PictureBody))) == NULL)
  DoError("load_pict: cannot allocate space",NULL);

 if ((fp = fopen(fn, "r")) == NULL)
  DoError("load_pict: error opening input file %s\n",fn);

/* WARNING - this fread is VERY unsafe! If assumes that the C compiler
 * puts all fields of a structure adjacent. This is not always the case.
 * It appears that it works with gcc on a sparcstation, but may not work
 * on other systems. */
 fread(&header,sizeof(struct rasterfile),1,fp);
 if (header.ras_magic != RAS_MAGIC)
  DoError("load_pict: only supports rasterfile format\n",NULL);
 if ((header.ras_type != RT_STANDARD)||
   (header.ras_maptype != RMT_NONE) ||
   (header.ras_maplength != 0))
  DoError("load_pict: unsupported rasterfile format\n",NULL);

 pict->width = header.ras_width;
 pict->height = header.ras_height;
 pict->depth = header.ras_depth;

 if (pict->depth == 32)
  uchar_width = pict->width * 4;
 else if (pict->depth == 8)
  uchar_width = ROUND2(pict->width);
 else if (pict->depth == 1)
  uchar_width = ROUND16(pict->width) >> 3;
 else
  DoError("load_pict: only depths of 1, 8, and 32 are supported\n",NULL);
 pict->uchar_width = uchar_width;

 pict->data = (UCHAR *) calloc(uchar_width * pict->height, sizeof(UCHAR));
 if (pict->data == NULL)
  DoError("load_pict: cannot allocate space\n",NULL);

 fread(pict->data, sizeof(UCHAR), uchar_width*pict->height, fp);
 fclose(fp);
 return pict;
}

Picture load_header(FILE *fp)
{
 Picture pict;
 int uchar_width;
 struct rasterfile header;

 if ((pict = (Picture)calloc(1,sizeof(PictureBody))) == NULL)
  DoError("load_header: cannot allocate space",NULL);

/* WARNING - this fread is VERY unsafe! If assumes that the C compiler
 * puts all fields of a structure adjacent. This is not always the case.
 * It appears that it works with gcc on a sparcstation, but may not work
 * on other systems. */
```

```
  if (fread(&header,sizeof(struct rasterfile),1,fp) != 1)
   DoError("load_header: error reading header",NULL);
  if (header.ras_magic != RAS_MAGIC)
   DoError("load_pict: only supports rasterfile format\n",NULL);
  if ((header.ras_type != RT_STANDARD)||
    (header.ras_maptype != RMT_NONE) ||
    (header.ras_maplength != 0))
   DoError("load_pict: unsupported rasterfile format\n",NULL);

  pict->width = header.ras_width;
  pict->height = header.ras_height;
  pict->depth = header.ras_depth;

  if (pict->depth == 32)
   uchar_width = pict->width * 4;
  else if (pict->depth == 8)
   uchar_width = ROUND2(pict->width);
  else if (pict->depth == 1)
   uchar_width = ROUND16(pict->width) >> 3;
  else
   DoError("load_header: only depths of 1, 8, and 32 are supported\n",NULL);
  pict->uchar_width = uchar_width;
  pict->data = NULL;

  return pict;
}

void write_pict(fn, pict)
char *fn;
Picture pict;
{
 FILE *fp;
 int uchar_width;
 struct rasterfile header;

 if ((fp = fopen(fn, "w")) == NULL)
  DoError("write_pict: error opening output file %s\n",fn);

 header.ras_magic = RAS_MAGIC;
 header.ras_width = pict->width;
 header.ras_height = pict->height;
 header.ras_depth = pict->depth;
 header.ras_length = pict->uchar_width*pict->height;
 header.ras_type = RT_STANDARD;
 if (pict->cmap == NULL) {
  header.ras_maptype = RMT_NONE;
  header.ras_maplength = 0;
/* WARNING - this fwrite is VERY unsafe! If assumes that the C compiler
 * puts all fields of a structure adjacent. This is not always the case.
 * It appears that it works with gcc on a sparcstation, but may not work
 * on other systems. */
  if (fwrite(&header,sizeof(struct rasterfile),1,fp) != 1)
   DoError("write_pict: error writing header",NULL);
 }
 else {
```

```
    header.ras_maptype = RMT_EQUAL_RGB;
    header.ras_maplength = pict->cmap->numberOfEntries*3;
/* WARNING - this fwrite is VERY unsafe! If assumes that the C compiler
 * puts all fields of a structure adjacent. This is not always the case.
 * It appears that it works with gcc on a sparcstation, but may not work
 * on other systems. */
    if (fwrite(&header,sizeof(struct rasterfile),1,fp) != 1)
      DoError("write_pict: error writing header",NULL);
    fwrite(pict->cmap->red,sizeof(UCHAR),pict->cmap->numberOfEntries,fp);
    fwrite(pict->cmap->green,sizeof(UCHAR),pict->cmap->numberOfEntries,fp);
    fwrite(pict->cmap->blue,sizeof(UCHAR),pict->cmap->numberOfEntries,fp);
  }

  uchar_width = pict->uchar_width;
  fwrite(pict->data, sizeof(UCHAR), uchar_width*pict->height, fp);
  fclose(fp);
}

void write_header(FILE *fp, Picture pict)
{
  struct rasterfile header;

  header.ras_magic = RAS_MAGIC;
  header.ras_width = pict->width;
  header.ras_height = pict->height;
  header.ras_depth = pict->depth;
  header.ras_length = pict->uchar_width*pict->height;
  header.ras_type = RT_STANDARD;
  header.ras_maptype = RMT_NONE;
  header.ras_maplength = 0;
/* WARNING - this fwrite is VERY unsafe! If assumes that the C compiler
 * puts all fields of a structure adjacent. This is not always the case.
 * It appears that it works with gcc on a sparcstation, but may not work
 * on other systems. */
  fwrite(&header,sizeof(struct rasterfile),1,fp);
}

#define BytesPerScanline(pict) (pict->uchar_width)

UCHAR ReadPixel(pict,x,y)
Picture pict;
int x,y;
{
  if (pict->depth == 8)
    return *(pict->data+y*BytesPerScanline(pict)+x);
  else if (pict->depth == 1)
    return ((*(pict->data+y*BytesPerScanline(pict)+(x>>3))) &
            bitmasks[x%8])?1:0;
  else
    DoError("ReadPixel: only depths of 1 and 8 are supported\n",NULL);
}

void WritePixel(pict,x,y,color)
Picture pict;
int x,y;
```

```
UCHAR color;
{
 if (x<0||x>=pict->width||y<0||y>=pict->height) {
  char s[256];
  sprintf(s,"%d %d",x,y);
  DoError("WritePixel: Out of bounds: ",s);
 }
 if (pict->depth == 8)
  *(pict->data+y*pict->uchar_width+x) = color;
 else if (pict->depth == 1)
  if (color)
   *(pict->data+y*BytesPerScanline(pict)+(x>>3)) |= bitmasks[x%8];
  else
   *(pict->data+y*BytesPerScanline(pict)+(x>>3)) &= ~bitmasks[x%8];
 else
  DoError("WritePixel: only depths of 1 and 8 are supported\n",NULL);
}

void WriteClippedPixel(pict,x,y,color)
Picture pict;
int x,y;
UCHAR color;
{
 if (x<0||x>=pict->width||y<0||y>=pict->height) {
  return;
 }
 if (pict->depth == 8)
  *(pict->data+y*pict->uchar_width+x) = color;
 else if (pict->depth == 1)
  if (color)
   *(pict->data+y*BytesPerScanline(pict)+(x>>3)) |= bitmasks[x%8];
  else
   *(pict->data+y*BytesPerScanline(pict)+(x>>3)) &= ~bitmasks[x%8];
 else
  DoError("WritePixel: only depths of 1 and 8 are supported\n",NULL);
}

void CopyPicture(Picture dest, Picture src)
{
 int uchar_width;
 dest->width = src->width;
 dest->height = src->height;
 dest->depth = src->depth;
 dest->uchar_width = BytesPerScanline(src);
 uchar_width = BytesPerScanline(src);
 memcpy(dest->data,src->data,uchar_width*src->height);
}
```

Jul 26 13:15 1991 read.c

```
#include <stdio.h>
#include "misc.h"
#include "read.h"

#define MAX_STRING_LEN (255)

int ReadInt(FILE *fp)
{
 char s[MAX_STRING_LEN];
 int x;

 fgets(s,MAX_STRING_LEN,fp);
 while (sscanf(s,"%d",&x)!=1)
  fprintf(stderr,"ReadInt: integer expected - reenter.\n");
 return x;
}

int ReadFloat(FILE *fp)
{
 char s[MAX_STRING_LEN];
 float x;

 fgets(s,MAX_STRING_LEN,fp);
 while (sscanf(s,"%f",&x)!=1)
  fprintf(stderr,"ReadFloat: integer expected - reenter.\n");
 return x;
}

char *ReadString(FILE *fp)
{
 char s[MAX_STRING_LEN];
 char *endPtr;

 fgets(s,MAX_STRING_LEN,fp);
 endPtr = strchr(s,'\n');
 if (endPtr != NULL)
  *endPtr = '\0';
 return strdup(s);
}
```

Aug 13 00:13 1991 Makefile

```
CCFLAGS = -g -c -I/net/piglet/piglet-1c/hopcroft/new/include

EXTRNS = /net/piglet/piglet-1c/hopcroft/error/error.o\
/net/piglet/piglet-1c/hopcroft/new/pict/pict.o \
/net/piglet/piglet-1c/hopcroft/lists/lists.o

ARGS_MODULE = /net/piglet/piglet-1c/hopcroft/new/ScanArgs/args.o

SOURCES = Makefile diff2.c dmain.c l2Norm2.c match.c matchparallel.c single.c
EXTRNSOURCES = /net/piglet/piglet-1c/hopcroft/error/error.c \
/net/piglet/piglet-1c/hopcroft/new/pict/pict.c \
/net/piglet/piglet-1c/hopcroft/lists/lists.c


INCLUDE = /net/piglet/piglet-1c/hopcroft/new/include/
ARGS = $(INCLUDE)args.h
BASELINES = $(INCLUDE)baselines.h
BLOBIFY = $(INCLUDE)blobify.h
BOOLEAN = $(INCLUDE)boolean.h
BOXES = $(INCLUDE)boxes.h
CONTOUR = $(INCLUDE)newContour.h
DESCRIPTORS = $(INCLUDE)descriptors.h
DICT = $(INCLUDE)dict.h
DIFF = $(INCLUDE)diff.h
DIFF2 = $(INCLUDE)diff2.h
ERROR = $(INCLUDE)error.h
LINES = $(INCLUDE)lines.h
LISTS = $(INCLUDE)lists.h
MATCH = $(INCLUDE)match.h
MATCHPARALLEL = $(INCLUDE)matchparallel.h
MISC = $(INCLUDE)misc.h
MYLIB = $(INCLUDE)mylib.h
NEWMATCH = $(INCLUDE)newMatch.h
ORIENT = $(INCLUDE)orient.h
PICT = $(INCLUDE)pict.h
READ = $(INCLUDE)read.h
TYPES = $(INCLUDE)types.h

INCSOURCES = $(BASELINES) $(BLOBIFY) $(BOOLEAN) $(BOXES) $(CONTOUR) \
$(DICT) $(DIFF) $(DIFF2) $(LINES) $(LISTS) $(MATCH) $(MATCHPARALLEL) \
$(ORIENT) $(PICT) $(TYPES)

anomalies:	anomalies.o diff2.o newMatch.o ../main/dict.o
	gcc anomalies.o diff2.o newMatch.o ../main/dict.o $(EXTRNS) -lm -o $@

descriptors:	descMain.o descriptors.o diff2.o newMatch.o newL2.o ../main/dict.o
	gcc descMain.o descriptors.o diff2.o newMatch.o newL2.o ../main/dict.o ../lib/mylib.a
	-lm -o $@

drawBlobs:	drawBlobs.o ../main/dict.o
	gcc drawBlobs.o ../main/dict.o ../lib/mylib.a -lm -o $@
```

```
compare:	diff2.o dmain.o newMatch.o ../main/dict.o
	gcc dmain.o diff2.o newMatch.o ../main/dict.o \
$(EXTRNS) -lm -o $@

equiv: equiv.o descriptors.o diff2.o newMatch.o newL2.o ../main/dict.o
	gcc equiv.o descriptors.o diff2.o newMatch.o newL2.o ../main/dict.o ../lib/mylib.a -lm -o $@

extract:	extract.o ../main/dict.o
	gcc extract.o ../main/dict.o $(EXTRNS) -o $@

l2Norm:		l2Norm2.o ../main/dict.o
	gcc l2Norm2.o ../main/dict.o $(EXTRNS) -lm -o $@

recogDesc:	recogDesc.o ../main/dict.o diff2.o newMatch.o newL2.o
	gcc recogDesc.o ../main/dict.o diff2.o newMatch.o newL2.o ../lib/mylib.a -lm -o $@

resample:	resample.o ../main/dict.o
	gcc resample.o ../main/dict.o $(EXTRNS) -lm -o $@

single:		single.o newMatch.o diff2.o newL2.o ../main/dict.o
	gcc single.o newMatch.o diff2.o newL2.o ../main/dict.o ../lib/mylib.a -lm -o $@

sortMatrix:	sortMatrix.o
	gcc sortMatrix.o $(EXTRNS) -o $@

printAll:	printIncludes printExtrns printCode

printCode:	$(SOURCES)
	/usr/5bin/pr -n3 $(SOURCES) | lpr -PWeeklyWorldNews

printExtrns:	$(EXTRNSOURCES)
	/usr/5bin/pr -n3 $(EXTRNSOURCES) | lpr -PWeeklyWorldNews

printIncludes:	$(INCSOURCES)
	/usr/5bin/pr -n3 $(INCSOURCES) | lpr -PWeeklyWorldNews

anomalies.o:	anomalies.c $(ERROR) $(TYPES) $(PICT) $(DICT) $(DIFF) $(MISC)
	gcc $(CCFLAGS) anomalies.c

descriptors.o:	descriptors.c $(MYLIB) $(TYPES) $(DICT) $(DIFF) $(MISC) $(DESCRIPTORS)
	gcc $(CCFLAGS) descriptors.c

descMain.o: descMain.c $(MYLIB) $(TYPES) $(DICT) $(DIFF) $(DESCRIPTORS)
	gcc $(CCFLAGS) descMain.c

diff2.o: diff2.c $(BOOLEAN) $(TYPES) $(PICT) $(DIFF2) $(NEWMATCH)
	gcc $(CCFLAGS) diff2.c

dmain.o: dmain.c $(BOOLEAN) $(PICT) $(DIFF)
	gcc $(CCFLAGS) dmain.c

drawBlobs.o:	drawBlobs.c $(MYLIB) $(TYPES) $(DICT)
	gcc $(CCFLAGS) drawBlobs.c
```

```
equiv.o:	equiv.c $(MYLIB) $(TYPES) $(DICT) $(DIFF) $(DESCRIPTORS)
	gcc $(CCFLAGS) equiv.c

extract.o:	extract.c $(BOOLEAN) $(TYPES) $(DICT)
	gcc $(CCFLAGS) extract.c

l2Norm2.o:	l2Norm2.c $(BOOLEAN) $(TYPES) $(ERROR) $(DICT)
	gcc $(CCFLAGS) l2Norm2.c

match.o: match.c $(BOOLEAN) $(TYPES) $(PICT) $(DIFF2) $(MATCH) $(MATCHPARALLEL)
	gcc $(CCFLAGS) match.c

matchparallel.o: matchparallel.c $(BOOLEAN) $(TYPES) $(PICT) $(DIFF2) \
	$(MATCH) $(MATCHPARALLEL)
	gcc $(CCFLAGS) matchparallel.c

newL2.o:	newL2.c $(BOOLEAN) $(ERROR) $(TYPES) $(DICT)
	gcc $(CCFLAGS) newL2.c

newMatch.o: newMatch.c $(ERROR) $(MISC) $(NEWMATCH) $(DICT) $(TYPES)
	gcc $(CCFLAGS) newMatch.c

recogDesc.o:	recogDesc.c $(MYLIB) $(TYPES) $(DICT) $(DIFF)
	gcc $(CCFLAGS) recogDesc.c

resample.o:	resample.c $(BOOLEAN) $(TYPES) $(ERROR) $(DICT)
	gcc $(CCFLAGS) resample.c

single.o:	single.c $(MYLIB) $(TYPES) $(DICT) $(DIFF) $(DIFF2) $(MATCH)
		$(MATCHPARALLEL)
	gcc $(CCFLAGS) single.c

sortMatrix.o:	sortMatrix.c $(ERROR) $(PICT)
	gcc $(CCFLAGS) sortMatrix.c
```

Jul 9 19:36 1991 anomalies.c

```
#include <stdio.h>
#include "error.h"
#include "types.h"
#include "pict.h"
#include "dict.h"
#include "diff.h"
#include "misc.h"

#define MAX_STRING_LEN (100)
#define MAX_DICTIONARIES (15)
#define MAX_WORDS (100)
#define MAX_ENTRIES (MAX_WORDS*MAX_WORDS)

typedef struct {
 float score;
 int x;
 int y;
} *CompareTuple,CompareTupleBody;


int ReadInt(FILE *fp)
{
 char s[MAX_STRING_LEN];
 int x;

 fgets(s,MAX_STRING_LEN,fp);
 while (sscanf(s,"%d",&x)!=1)
  fprintf(stderr,"ReadInt: integer expected - reenter.\n");
 return x;
}

char *ReadString(FILE *fp)
{
 char s[MAX_STRING_LEN];
 char *endPtr;

 fgets(s,MAX_STRING_LEN,fp);
 endPtr = strchr(s,'\n');
 if (endPtr != NULL)
  *endPtr = '\0';
 return strdup(s);
}

int TupleLessThan(CompareTuple *x,CompareTuple *y)
{
 if ((*x)->score == (*y)->score)
  return 0;
 else if ((*x)->score < (*y)->score)
  return -1;
 else
  return 1;
}
```

```
int CountAnomalies(Dictionary d1, Dictionary d2,char *dName1,char *dName2,char **words,FILE *outfp)
{
 CompareTupleBody scoreBodies[MAX_ENTRIES];
 CompareTuple scores[MAX_ENTRIES];
 Picture pict;
 int x,y,i,j;
 int anomalies;

 pict = CompareDictionaries(d1,d2);

 for (y=0,i=0;y<pict->height; ++y)
  for (x=0;x<pict->width; ++x) {
   CompareTuple temp;
/*    temp = (CompareTuple)calloc(1,sizeof(CompareTupleBody));
   if (temp==NULL)
       DoError("%s: cannot allocate space.\n",argv[0]);
*/
   temp = scoreBodies+i;
   temp->score = *((float *)(pict->data)+x+y*pict->width);
   temp->x = x;
   temp->y = y;
   scores[i] = temp;
   ++i;
  }
 qsort(scores,i,sizeof(CompareTuple),TupleLessThan);
 for (j=0,anomalies=0;j<d1->numberOfEntries; ++j)
  if (scores[j]->x != scores[j]->y) {
   fprintf(outfp,"%s:%s  %s:%s\n",dName1,words[scores[j]->x],
                    dName2,words[scores[j]->y]);
   ++anomalies;
  }

 free_pict(pict);
 return anomalies;
}

void main(int argc,char **argv)
{
 char *outFile, *listFile;
 int numberOfDictionaries;
 Dictionary dictionaries[MAX_DICTIONARIES];
 char *names[MAX_DICTIONARIES];
 char *words[MAX_WORDS];
 int numberOfWords;
 FILE *listfp,*outfp;
 int anomalies[MAX_DICTIONARIES][MAX_DICTIONARIES];
 int i,x,y;

 if (argc != 3)
  DoError("Usage: %s listfile outfile.\n",argv[0]);
 listFile = argv[1];
 outFile = argv[2];

```

```
 if ((listfp = fopen(listFile,"r"))==NULL)
  DoError("Error opening file %s.\n",listFile);

/* Read in the number of words in each dictionary */
 numberOfWords = ReadInt(listfp);
 if (numberOfWords > MAX_WORDS)
  DoError("%s: too many words.\n",argv[0]);

/* Read in the words */
 for (i=0;i<numberOfWords; ++i) {
  words[i] = ReadString(listfp);
 }

/* Read in the number of dictionaries */
 numberOfDictionaries = ReadInt(listfp);
 if (numberOfDictionaries > MAX_DICTIONARIES)
  DoError("%s: too many dicitionaries.\n",argv[0]);

/* Read in the dictionaries and their names */
 for (i=0;i<numberOfDictionaries; ++i) {
  names[i] = ReadString(listfp);
  dictionaries[i] = ReadDictionary(names[i]);
 }

/* Check to see that all dictionaries have the same number of shapes as the specified number of words. */
 for (i=1;i<numberOfDictionaries; ++i)
  if (dictionaries[i]->numberOfEntries != numberOfWords)
   DoError("Dictionary %s has wrong number of entries.\n",names[i]);

/* Write the results */
 if ((outfp = fopen(outFile,"w"))==NULL)
  DoError("Error opening %s for output.\n",outFile);
 fprintf(outfp,"Words:\n");
 for (i=0;i<numberOfWords; ++i)
  fprintf(outfp,"%d: %s\n",i,words[i]);
 fprintf(outfp,"\n");
 fprintf(outfp,"Dictionaries:\n");
 for (i=0;i<numberOfDictionaries; ++i)
  fprintf(outfp,"%d: %s\n",i,names[i]);
 fprintf(outfp,"\n");

/* Fill in the anomaly counts */
 for (y=0;y<numberOfDictionaries; ++y)
  for (x=0;x<numberOfDictionaries; ++x) {
   anomalies[y][x] =
CountAnomalies(dictionaries[y],dictionaries[x],names[y],names[x],words,outfp);
   printf("(%d,%d): %d\n",x,y,anomalies[y][x]);
  }

 fprintf(outfp,"\n\n");
 fprintf(outfp,"   ");
 for (x = 0; x < numberOfDictionaries; x++)
  fprintf(outfp,"%7d ", x);
 fprintf(outfp, "\n");
```

```
for (y=0;y<numberOfDictionaries;++y) {
 fprintf(outfp, " %3d ", y);
 for (x=0;x<numberOfDictionaries;++x)
  fprintf(outfp,"%7d ",anomalies[y][x]);
 fprintf(outfp, "\n");
}
fclose(outfp);

}
```

Jul 31 17:14 1991 descMain.c

```
#include <stdio.h>
#include "mylib.h"
#include "types.h"
#include "dict.h"
#include "diff.h"
#include "descriptors.h"

void PrintDescriptors(Dictionary models,char *modelName,char **wordNames,
                int numberOfFonts,Dictionary fonts[],
                char **fontNames,int numberOfWords,
                DiffDescriptor dd)
{
 int modelIndex;fontIndex;
 int starCount,correctCount;
 Descriptor thisDescriptor;
 int lineCount;

 printf("\f\n");
 PrintWords(wordNames,numberOfWords);
 lineCount = 0;
 starCount = 0;
 correctCount = 0;
 for (modelIndex=0;modelIndex<numberOfWords; ++modelIndex) {
  printf("%s %s\n",modelName,wordNames[modelIndex]);
  ++lineCount;
  for (fontIndex=0;fontIndex<numberOfFonts; ++fontIndex) {
   thisDescriptor =
ComputeDescriptor(modelIndex,models,fonts[fontIndex],numberOfWords,dd);
   printf("  ");
   PrintField(fontNames[fontIndex],20);
   PrintDescriptor(thisDescriptor,&starCount,&correctCount);
   printf("\n");
   ++lineCount;
  }
  if (lineCount>30) {
   printf("\f\n");
   PrintWords(wordNames,numberOfWords);
   lineCount = 0;
  }
 }
 fprintf(stdout,"There were %d mismatches ",starCount-
numberOfWords*numberOfFonts);
 fprintf(stdout,"better than %d correct matches. (%6.2f%)\n",
        numberOfWords*numberOfFonts,
        (float)(numberOfWords*numberOfFonts)/(float)starCount);
 fprintf(stdout,"There were %d correctly matched words out of %d. (%6.2f%)\n",
        correctCount,numberOfWords*numberOfFonts,
        (float)correctCount/(float)numberOfWords/numberOfFonts);
}

void main(int argc,char **argv)
{
```

```
 char *listFile;
 Dictionary models;
 char *modelName;
 int numberOfFonts;
 Dictionary fonts[MAX_FONTS];
 char *fontNames[MAX_FONTS];
 char *wordNames[MAX_WORDS];
 int numberOfWords;
 float centerWeight;
 int normalBandWidth;
 BOOLEAN
lengthNormalize,useL2,slopeConstrain,warp,topToBottomOption,hillToValleyOption;
 BOOLEAN separate;
 float topToBottom,hillToValleyLocal;
 FILE *listfp;
 int i,x,y;
 DiffDescriptorBody dd;

 centerWeight = 1.0;
 normalBandWidth = 20;
 topToBottom = 1.0;
 hillToValleyLocal = 1.0;
 DefArg("%s","listFile",&listFile);
 DefOption("-L2","-L2",&useL2);
 DefOption("-slopeConstrain %f","-slopeConstrain <center weight>",
         &slopeConstrain,&centerWeight);
 DefOption("-warp %f %d","-warp <center weight> <band width>",
         &warp,&centerWeight,&normalBandWidth);
 DefOption("-separate","-separate",&separate);
 DefOption("-normalize","-normalize",&lengthNormalize);
 DefOption("-topToBottom %f","-topToBottom
<ratio>",&topToBottomOption,&topToBottom);
 DefOption("-hillToValley %f","-hillToValley
<ratio>",&hillToValleyOption,&hillToValleyLocal);
 ScanArgs(argc,argv);

 if ((listfp = fopen(listFile,"r"))==NULL)
  DoError("Error opening file %s.\n",listFile);

/* Read in the number of words in each dictionary */
 numberOfWords = ReadInt(listfp);
 if (numberOfWords > MAX_WORDS)
  DoError("%s: too many words.\n",argv[0]);

/* Read in the words */
 for (i=0;i<numberOfWords;++i) {
  wordNames[i] = ReadString(listfp);
 }

/* Read in the model dictionary */
 modelName = ReadString(listfp);
 models = ReadDictionary(modelName);

/* Read in the number of dictionaries */
 numberOfFonts = ReadInt(listfp);
```

```
  if (numberOfFonts > MAX_FONTS)
   DoError("%s: too many dictionaries.\n",argv[0]);

 /* Read in the dictionaries and their names */
  for (i=0;i<numberOfFonts;++i) {
   fontNames[i] = ReadString(listfp);
   fonts[i] = ReadDictionary(fontNames[i]);
  }

 /* Check to see that all dictionaries have the same number of shapes as the specified number
 of words. */
  for (i=1;i<numberOfFonts;++i)
   if (fonts[i]->numberOfEntries < numberOfWords)
    DoError("Dictionary %s has too few entries.\n",fontNames[i]);
  if (models->numberOfEntries < numberOfWords)
   DoError("Model dictionary has too few of entries.\n",NULL);


  if (useL2) {
   fprintf(stdout,"Using L2 on length normalized shapes.\n");
   dd.diffType = L2;
  }
  else if (slopeConstrain) {
   fprintf(stdout,"Using dynamic time warping with slope contrained to [0.5,2].\n");
   dd.diffType = CONSTRAINED;
   dd.separate = separate;
   if (separate)
    fprintf(stdout,"Top and bottom warped separately.\n");
   else
    fprintf(stdout,"Top and bottom warped together.\n");
  }
  else {
   fprintf(stdout,"Using dynamic time warping with bandwidth %d.\n",normalBandWidth);
   dd.diffType = WARP;
   dd.bandWidth = normalBandWidth;
   dd.separate = separate;
   if (separate)
    fprintf(stdout,"Top and bottom warped separately.\n");
   else
    fprintf(stdout,"Top and bottom warped together.\n");
  }
  if (!useL2) {
   fprintf(stdout,"Center weight = %f.\n",centerWeight);
   dd.centerWeight = centerWeight;
   if (lengthNormalize) {
    dd.lengthNormalize = TRUE;
    fprintf(stdout,"Scores normalized by signal length.\n");
   }
   else
    dd.lengthNormalize = FALSE;
  }
  dd.hillToValley = hillToValleyLocal;
  dd.topToBottom = topToBottom;
  dd.pathFP = NULL;
```

```
 fprintf(stdout,"Words:\n");
 for (i=0;i<numberOfWords; ++i)
  fprintf(stdout,"%d: %s\n",i,wordNames[i]);
 fprintf(stdout,"\n");
 fprintf(stdout,"Model font is %s.\n",modelName);
 fprintf(stdout,"Fonts:\n");
 for (i=0;i<numberOfFonts; ++i)
  fprintf(stdout,"%d: %s\n",i,fontNames[i]);
 fprintf(stdout,"\n");


PrintDescriptors(models,modelName,wordNames,numberOfFonts,fonts,fontNames,numberOfWords,&dd);
}
```

Jul 26 13:48 1991 descriptors.c

```
#include <stdio.h>
#include "mylib.h"
#include "types.h"
#include "dict.h"
#include "diff.h"
#include "misc.h"
#include "descriptors.h"

typedef struct {
 float score;
 int word;
} *CompareTuple,CompareTupleBody;

int TupleLessThan(CompareTuple x,CompareTuple y)
{
 if (x->score == y->score)
  return 0;
 else if (x->score < y->score)
  return -1;
 else
  return 1;
}

int CompareDescriptorElements(Descriptor x,Descriptor y)
{
 if (*x == *y)
  return 0;
 else if (*x < *y)
  return -1;
 else
  return 1;
}

Descriptor ComputeDescriptor(int modelIndex,Dictionary models,Dictionary thisFont,int
numberOfWords,
                         DiffDescriptor dd)
{
 DescriptorElement descriptor[MAX_WORDS+1];
 CompareTupleBody results[MAX_WORDS];
 int i;

 for (i=0;i<numberOfWords; ++i) {
  results[i].score =
DiffPair(*(models->outlines+modelIndex),*(thisFont->outlines+i),dd);
  results[i].word = i;
 }
 qsort(results,thisFont->numberOfEntries,sizeof(CompareTupleBody),TupleLessThan);
 for (i=0;i<numberOfWords; ++i) {
  descriptor[i] = results[i].word+1; /* Descriptor values are one greater than word indices
*/
  if (results[i].word == modelIndex) {
   ++i;
```

```
      break;
     }
    }
    descriptor[l] = '\0';
    qsort(descriptor,i,sizeof(DescriptorElement),CompareDescriptorElements);
    return (Descriptor)strdup((char *)descriptor);
   }

   void PrintField(char *s,int w)
   {
    int i,l;
    printf("%s",s);
    l = w-strlen(s);
    for (i=0;i<l;++i)
     printf(" ");
   }

   void PrintDescriptor(Descriptor d,int *starCount,int *correctCount)
   {
    int i=1;        /* Descriptor values are one greater than word indices */
    int temp;
    temp = *starCount;
    if (*d == '\0') {
     printf("*");
     ++*starCount;
    }
    while (*d != '\0') {
     while (i++ < *d)
      printf(" ");
     printf("*");
     ++*starCount;
     d++;
    }
    if (*starCount-temp == 1)
     ++*correctCount;
   }

   void PrintWords(char **words,int numberOfWords)
   {
    int lengths[MAX_WORDS];
    int i,j;
    int maxLength = 0;

    maxLength = 0;
    for (i=0;i<numberOfWords;++i) {
     lengths[i] = strlen(words[i]);
     if (lengths[i] > maxLength)
      maxLength = lengths[i];
    }

    for (j=0;j<maxLength;++j) {
     printf("               ");
     for (i=0;i<numberOfWords;++i)
      if (j<lengths[i])
         printf("%c",*(words[i]+j));
```

```
       else
           printf(" ");
     printf("\n");
    }
   }
```

Jul 31 17:11 1991 diff2.c

```
#include <stdio.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"
#include "diff.h"
#include "newMatch.h"


extern double fabs(double);

/* Dynamic programming version of DiffPair */
inline float DiffPair(OutlinePair one, OutlinePair two,
                DiffDescriptor dd)
{
 hillToValley = dd->hillToValley;
 if ((dd->separate)&&(dd->pathFP!=NULL))
  DoError("DiffPair: separate cannot be used with pathfile option.\n",NULL);
 switch (dd->diffType) {
 case CONSTRAINED:
  if (dd->pathFP != NULL)
   return SlopeCMatchAndPath(one->top,one->bottom,one->numberOfLegs,
                     two->top,two->bottom,two->numberOfLegs,
                     dd->centerWeight,dd->lengthNormalize,dd->topToBottom,
                     dd->pathFP);
  else
   if (dd->separate)
       return
       SepSlopeCMatch(one->top,one->numberOfLegs,two->top,two->numberOfLegs,
                   dd->centerWeight,dd->lengthNormalize)*dd->topToBottom +

       SepSlopeCMatch(one->bottom,one->numberOfLegs,two->bottom,two->number
       OfLegs,
                   dd->centerWeight,dd->lengthNormalize);
   else
       return SlopeCMatch(one->top,one->bottom,one->numberOfLegs,
                 two->top,two->bottom,two->numberOfLegs,
                 dd->centerWeight,dd->lengthNormalize,dd->topToBottom);
  break;
 case L2:
  if (dd->pathFP != NULL)
   DoError("DiffPair: L2 does not support path computation.\n",NULL);
  else
   return L2Compare(one,two,dd->topToBottom);
  break;
 case WARP:
  if (dd->pathFP != NULL)
   return NewMatchAndPath(one->top,one->bottom,one->numberOfLegs,
                 two->top,two->bottom,two->numberOfLegs,
                 dd->centerWeight,dd->lengthNormalize,dd->bandWidth,
                 dd->topToBottom,
```

```
                        dd->pathFP);
   else
    if (dd->separate)
        return SepMatch(one->top,one->numberOfLegs,two->top,two->numberOfLegs,

               dd->centerWeight,dd->lengthNormalize,dd->bandWidth)*dd->topToBotto
               m +

        SepMatch(one->bottom,one->numberOfLegs,two->bottom,two->numberOfLegs,
                       dd->centerWeight,dd->lengthNormalize,dd->bandWidth);
    else
        return
        NewMatch(one->top,one->bottom,one->numberOfLegs,two->top,two->bottom,
        two->numberOfLegs,
                      dd->centerWeight,dd->lengthNormalize,dd->bandWidth,
                      dd->topToBottom);
   break;
  default:
   DoError("DiffPair: internal error.\n",NULL);
  }
 }

 #ifdef foo
 inline float DiffPairAndPath(OutlinePair one, OutlinePair two,
                        float centerWeight,BOOLEAN lengthNormalize,int
                      normalBandWidth,
                        char *filename,BOOLEAN doPath)
 {
  FILE *fp;
  float score;
  if ((fp=fopen(filename,"w"))==NULL)
   DoError("DiffPairAndMatch: error opening output file %s.\n",filename);
  score = NewMatchAndPath(one->top,one->bottom,one->numberOfLegs,
                      two->top,two->bottom,two->numberOfLegs,
                      centerWeight,lengthNormalize,normalBandWidth,
                      fp,doPath);
  fclose(fp);
  return score;
 }
 #endif

 BOOLEAN IsSymmetric(Picture pict)
 {
  int x,y;
  float maxDiff = 0;
  for (y=0;y<pict->height; ++y)
   for (x=0;x<pict->width; ++x) {
    float temp =  fabs (*((float *)(pict->data)+pict->width*y+x) -
                      *((float *)(pict->data)+pict->width*x+y));
    if (temp > maxDiff)
        maxDiff = temp;
   }
  fprintf(stderr,"maxDiff = %f.\n",maxDiff);
  if (maxDiff > 0.01)
   return FALSE;
```

```
    return TRUE;
   }

   /* Given the names of two dictionary files, compute the squared difference
    * between every pair of shapes in the cross product of the dictionaries.
    * The result is a matrix printed to stdout. The width and height are
    * followed by the matrix entries in row major order. The output is in
    * ascii to facilitate reading by a Symbolics. */
   Picture CompareDictionaries(Dictionary dict1, Dictionary dict2,DiffDescriptor dd)
   {
    Picture pict;
    int x,y;
    pict = new_pict(dict2->numberOfEntries,
                    dict1->numberOfEntries,
                    32);

    for (y=0;y<pict->height;++y)
     for (x=0;x<pict->width;++x) {
      /* for output files when printing and match */
      printf("---> (%d,%d) <---\n",y,x);
      *((float *)(pict->data)+pict->width*y+x) =
          DiffPair(*(dict1->outlines+y),
                   *(dict2->outlines+x),
                   dd);
     }
    if (!IsSymmetric(pict))
     fprintf(stderr,"Matrix is not symmetric.\n");
    return pict;
   }

   void WritePictureAsAscii(Picture pict,char *filename,
                            char *info1, char *info2)
   {
    FILE *fp;
    int x,y;
    int count;

    if ((fp = fopen(filename,"w"))==NULL)
     DoError("WritePictureAsAscii: error opening output file\n",NULL);
    fprintf(fp,"%s\n",info1);
    fprintf(fp,"%s\n",info2);
    fprintf(fp,"#\n");
    fprintf(fp,"%d\n%d\n",pict->width,pict->height);
    fprintf(fp, "%3s ", "");
    for (x = 0; x < pict->width; x++)
      fprintf(fp,"%7d ", x);
    fprintf(fp, "\n");
    for (y=0;y<pict->height;++y) {
     fprintf(fp, " %3d ", y);
     count = 1;
     for (x=0;x<pict->width;++x) {
      fprintf(fp,"%7.3f ",*(((float*)pict->data)++));
   /*   if ((pict->width > 10) && (!((count++)%10)))
         fprintf(fp,"\n");
   */ }
```

```
    fprintf(fp, "\n");
   }
   fclose(fp);
  }
```

Jul 22 15:21 1991 dmain.c

```
#include <stdio.h>
#include <math.h>
#include <values.h>
#include "boolean.h"
#include "types.h"
#include "pict.h"
#include "dict.h"
#include "diff.h"
#include "match.h"
#include "matchparallel.h"


void main(int argc,char **argv)
{
 Picture pict;
 char *infile1,*infile2,*outfile,*format;
 Dictionary dict1,dict2;

 if (argc != 5) {
  printf("Usage:\n");
  printf("  %s infile1 infile2 outfile format\n",argv[0]);
  printf("  where format is either ascii or pict.\n");
  exit(-1);
 }

 infile1 = argv[1];
 infile2 = argv[2];
 outfile = argv[3];
 format = argv[4];

 dict1 = ReadDictionary(infile1);
 dict2 = ReadDictionary(infile2);
 pict = CompareDictionaries(dict1,dict2,1,TRUE,20,FALSE);
 if (!strcmp(format,"pict"))
  write_pict(outfile,pict);
 else
  WritePictureAsAscii(pict,outfile,dict1->infoString,dict2->infoString);
}
```

Aug 15 20:20 1991 drawBlobs.c

```
#include <stdio.h>
#include "mylib.h"
#include "types.h"
#include "dict.h"

#define WIDTH (800)
#define H_MARGIN (20)
#define V_MARGIN (60)
#define H_SPACING (20)
#define V_SPACING (150) /* Must be greater than 2*X_HEIGHT */
#define X_HEIGHT (17)

extern int irint(double);

void DrawVLine(Picture pict,int x,int yt,int yb)
{
 int i;
 for (i=yt;i<yb;++i)
  WritePixel(pict,x,i,1);
}

void DrawOutline(Picture pict,OutlinePair o,int x,int y)
{
 int i,top,bottom;
 for (i=0;i<o->numberOfLegs;++i) {
  top = irint(-*(o->top+i)*X_HEIGHT);
  bottom = irint(*(o->bottom+i)*X_HEIGHT+X_HEIGHT);
  DrawVLine(pict,i+x,top+y,bottom+y);
 }
}

int main(int argc,char **argv)
{
 char *infile,*outfile;
 Dictionary dict;
 Picture pict;
 int i,totalLegs,totalLines;
 int x,y,newX;

 DefArg("%s %s","infile outfile",&infile,&outfile);
 ScanArgs(argc,argv);

 dict = ReadDictionary(infile);

 for (i=0,totalLegs=H_MARGIN,totalLines=V_MARGIN;i<dict->numberOfEntries;++i) {
  OutlinePair thisOutline = *(dict->outlines+i);
  totalLegs += thisOutline->numberOfLegs + H_SPACING;
  if (totalLegs > WIDTH) {
   totalLines += V_SPACING;
   totalLegs = H_MARGIN + thisOutline->numberOfLegs + H_SPACING;
   if (totalLegs > WIDTH)
```

```
        DoError("%s: Shape is too wide.\n",argv[0]);
     }
   }

   pict = new_pict(WIDTH,totalLines+V_MARGIN*2,1);

   for (i=0,x=H_MARGIN,y=V_MARGIN;i<dict->numberOfEntries; ++i) {
    OutlinePair thisOutline = *(dict->outlines+i);

    newX = x + thisOutline->numberOfLegs + H_SPACING;
    if (newX > WIDTH) {
     newX = H_MARGIN+thisOutline->numberOfLegs+H_SPACING;
     x = H_MARGIN;
     y += V_SPACING;
    }

    DrawOutline(pict,*(dict->outlines+i),x,y);

    x = newX;
   }

   write_pict(outfile,pict);
 }
```

Jul 26 16:47 1991 equiv.c

```
#include <stdio.h>
#include "mylib.h"
#include "types.h"
#include "dict.h"
#include "diff.h"
#include "descriptors.h"

void PrintEquivalenceClasses(int numberOfWords,char **wordNames,
                        int numberOfFonts,Dictionary fonts[],char **fontNames,
                        DiffDescriptor dd)
{
 Descriptor descriptors[MAX_FONTS*MAX_FONTS];
 int matchesWord[MAX_WORDS];
 int word,font1,font2,i;
 int totalDifferent,totalWords;
 int numberOfPairs;

 for (word=0;word<numberOfWords;++word) {
  for (font1=0,numberOfPairs=0;font1<numberOfFonts;++font1)
   for (font2=0;font2<font1;++font2)
      descriptors[numberOfPairs++] = ComputeDescriptor(word,fonts[font1],
                                          fonts[font2],numberOfWords,dd);
  for (i=0;i<numberOfWords;++i)
   matchesWord[i] = 0;
  for (i=0;i<numberOfPairs;++i) {
   Descriptor p;
   p = descriptors[i];
   while (*p!='\0')
      matchesWord[*p++ - 1]++;
  }
  for (i=0,totalDifferent=0,totalWords=0;i<numberOfWords;++i) {
   if (matchesWord[i])
      ++totalDifferent;
   totalWords += matchesWord[i];
  }
  printf("%20s:\t\t%6d %6.2f %6d %6.2f\n",wordNames[word],totalWords,
         (float)totalWords/numberOfPairs,totalDifferent,
         (float)totalDifferent/(float)totalWords*numberOfPairs);
  fprintf(stderr,"%20s:\t\t%6d %6.2f %6d %6.2f\n",wordNames[word],totalWords,
         (float)totalWords/numberOfPairs,totalDifferent,
         (float)totalDifferent/(float)totalWords*numberOfPairs);
 }
}


void main(int argc,char **argv)
{
 char *listFile;
 int numberOfFonts;
 Dictionary fonts[MAX_FONTS];
 char *fontNames[MAX_FONTS];
 char *wordNames[MAX_WORDS];
```

```
 int numberOfWords;
 float centerWeight;
 int normalBandWidth;
 BOOLEAN lengthNormalize,useL2,slopeConstrain,warp,topToBottomOption,hillToValleyOption;
 float topToBottom,hillToValleyLocal;
 FILE *listfp;
 int i,x,y;
 DiffDescriptorBody dd;

 centerWeight = 1.0;
 normalBandWidth = 20;
 topToBottom = 1.0;
 hillToValleyLocal = 1.0;
 DefArg("%s","listFile",&listFile);
 DefOption("-L2","-L2",&useL2);
 DefOption("-slopeConstrain %f","-slopeConstrain <center weight>",
        &slopeConstrain,&centerWeight);
 DefOption("-warp %f %d","-warp <center weight> <band width>",
        &warp,&centerWeight,&normalBandWidth);
 DefOption("-normalize","-normalize",&lengthNormalize);
 DefOption("-topToBottom %f","-topToBottom <ratio>",&topToBottomOption,&topToBottom);
 DefOption("-hillToValley %f","-hillToValley <ratio>",&hillToValleyOption,&hillToValleyLocal);
 ScanArgs(argc,argv);

 if ((listfp = fopen(listFile,"r"))==NULL)
  DoError("Error opening file %s.\n",listFile);

/* Read in the number of words in each dictionary */
 numberOfWords = ReadInt(listfp);
 if (numberOfWords > MAX_WORDS)
  DoError("%s: too many words.\n",argv[0]);

/* Read in the words */
 for (i=0;i<numberOfWords; ++i) {
  wordNames[i] = ReadString(listfp);
 }

/* Read in the number of dictionaries */
 numberOfFonts = ReadInt(listfp);
 if (numberOfFonts > MAX_FONTS)
  DoError("%s: too many dictionaries.\n",argv[0]);

/* Read in the dictionaries and their names */
 for (i=0;i<numberOfFonts; ++i) {
  fontNames[i] = ReadString(listfp);
  fonts[i] = ReadDictionary(fontNames[i]);
 }

/* Check to see that all dictionaries have the same number of shapes as the specified number of words. */
 for (i=1;i<numberOfFonts; ++i)
  if (fonts[i]->numberOfEntries < numberOfWords)
```

```
    DoError("Dictionary %s has too few entries.\n",fontNames[i]);

  if (useL2) {
   printf("Using L2 on length normalized shapes.\n");
   dd.diffType = L2;
  }
  else if (slopeConstrain) {
   printf("Using dynamic time warping with slope contrained to [0.5,2].\n");
   dd.diffType = CONSTRAINED;
  }
  else {
   printf("Using dynamic time warping with bandwidth %d.\n",normalBandWidth);
   dd.diffType = WARP;
   dd.bandWidth = normalBandWidth;
  }
  if (!useL2) {
   printf("Center weight = %f.\n",centerWeight);
   dd.centerWeight = centerWeight;
   if (lengthNormalize) {
    dd.lengthNormalize = TRUE;
    printf("Scores normalized by signal length.\n");
   }
   else
    dd.lengthNormalize = FALSE;
  }
  dd.hillToValley = hillToValleyLocal;
  dd.topToBottom = topToBottom;
  dd.pathFP = NULL;

  printf("Fonts:\n");
  for (i=0;i<numberOfFonts;++i)
   printf("%d: %s\n",i,fontNames[i]);
  printf("\n");


PrintEquivalenceClasses(numberOfWords,wordNames,numberOfFonts,fonts,fontNames,&dd);
}
```

Jul 3 14:31 1991 extract.c

```
#include <stdio.h>
#include <math.h>
#include <values.h>
#include "boolean.h"
#include "types.h"
#include "dict.h"

#define MAX_STRING_LEN 100
int ReadInt(FILE *fp)
{
 char s[MAX_STRING_LEN];
 int x;

 fgets(s,MAX_STRING_LEN,fp);
 while (sscanf(s,"%d",&x)!=1)
  fprintf(stderr,"ReadInt: integer expected - reenter.\n");
 return x;
}

void main(int argc,char **argv)
{
 char *infile,*listFile,*outfile;
 Dictionary dict1,dict2;
 int i;
 int numberOfEntries;
 FILE *fp;

 if (argc != 4) {
  printf("Usage:\n");
  printf("  %s infile listfile outfile\n",argv[0]);
  exit(-1);
 }

 infile = argv[1];
 listFile = argv[2];
 outfile = argv[3];

 dict1 = ReadDictionary(infile);

 if ((fp=fopen(listFile,"r"))==NULL)
  DoError("%s: error reading list file.\n",argv[0]);

 numberOfEntries = ReadInt(fp);
 if (numberOfEntries < 0)
  DoError("%s: list file must have a positive number of elements.\n",argv[0]);
 printf("Copying %d shapes.\n",numberOfEntries);

 dict2 = NewDict(numberOfEntries);

 dict2->infoString = dict1->infoString;
 for (i=0;i<numberOfEntries;++i) {
  int shape;
```

```
    shape = ReadInt(fp);
    if ((shape<0)||(shape>=dict1->numberOfEntries))
      DoError("%s: bad shape index.\n",argv[0]);
    *(dict2->outlines+i) = *(dict1->outlines+shape);
    *(dict2->rawOutlines+i) = *(dict1->rawOutlines+shape);
  }
  fclose(fp);
  WriteDictionary(dict2,outfile);

}
```

Jun 14 16:12 1991 l2Norm.c

```
#include <stdio.h>
#include <values.h>
#include <string.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "dict.h"

float L2Norm(OutlinePair signal, int startOffset,
            OutlinePair model)
{
 float *top1,*top2,*bottom1,*bottom2;
 int i,overlap;
 float sum;
 float temp;

 if ((startOffset < 0) ||
   (startOffset + model->numberOfLegs > signal->numberOfLegs))
  DoError("L2Norm: the model must overlap the signal.\n",NULL);

 top1 = signal->top+startOffset;
 top2 = model->top;
 bottom1 = signal->bottom+startOffset;
 bottom2 = model->bottom;

 overlap = signal->numberOfLegs - startOffset;
 if (overlap > model->numberOfLegs)
  overlap = model->numberOfLegs;

 for (i=0,sum=0;i<overlap;++i) {
  temp = *top1++ - *top2++;
  sum += temp * temp;
  temp = *bottom1++ - *bottom2++;
  sum += temp * temp;
 }

 return sum;
}

OutlinePair LookupShape(char c, Dictionary models)
{
/* dictionary file has the following order:
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  abcdefghijklmnopqrstuvwxyz
  0123456789
  ,.
*/
 int shapeIndex;
 if ((c>='a'&&c<='z'))
  shapeIndex = c-'a';
 else if (c == ',')
  shapeIndex = 26;
```

```
  else if (c == '.')
    shapeIndex = 27;
  else
    DoError("LookupShape: have no shape one of the characters.\n",NULL);
  return *(models->outlines+shapeIndex);
}


#define MAX_STRING_LENGTH 30
#define MAX_SHIFT 10
#define MAX_OVERLAP 5
float L2CompareWithString(OutlinePair signal, char *string,
                          Dictionary models)
{
  float *costMatrix;
  int *pathMatrix;
  int numberOfChars;
  int letterIndex, startOffset;
  float *cursor;
  int left;
  int right;
  OutlinePair modelShapes[MAX_STRING_LENGTH];
  char *charCursor;
  float minValue;
  float temp;
  int i,oldLeft,oldRight,minIndex;;

/* Make sure input string is not too long. */
  numberOfChars = strlen(string);
  if (numberOfChars >= MAX_STRING_LENGTH)
    DoError("L2CompareWithString: string is too long.\n",NULL);

/* Allocate space for dynamic programming array. */
/* For now, be a space hog. */
  costMatrix = (float *)calloc(signal->numberOfLegs*numberOfChars,
                               sizeof(float));
  pathMatrix = (int *)calloc(signal->numberOfLegs*numberOfChars,
                             sizeof(int));
  if ((costMatrix == NULL)||(pathMatrix == NULL))
    DoError("L2CompareWithString: cannot allocate space.\n",NULL);

/* Lookup the shapes corresponding to the characters in the string. */
  charCursor = string;
  for (i=0;i<numberOfChars; ++i)
    modelShapes[i] = LookupShape(*charCursor++,models);

/* Since the cost matrix is larger than the region containing valid
 * alignments, first fill in the array with large costs.  Later, some
 * of these will be overwritten. */
  cursor = costMatrix;
  for (i=0;i<signal->numberOfLegs*numberOfChars; ++i)
    *cursor++ = MAXFLOAT;

/* Fill in leftmost column */
  left = 0;
```

```
    right = MAX_SHIFT;
    for (startOffset=left;startOffset<right; ++startOffset)
     if (startOffset + modelShapes[0]->numberOfLegs <=
          signal->numberOfLegs)
      *(costMatrix+startOffset*numberOfChars) =
          L2Norm(signal, startOffset, modelShapes[0]);

  /* Now do the rest of the columns */
   for (letterIndex = 1; letterIndex < numberOfChars; ++letterIndex) {
    oldLeft = left;
    oldRight = right;
    left += modelShapes[letterIndex-1]->numberOfLegs;
    right += modelShapes[letterIndex-1]->numberOfLegs + MAX_SHIFT;
    for (startOffset=left;startOffset<right; ++startOffset) {
     if (startOffset + modelShapes[letterIndex]->numberOfLegs <=
           signal->numberOfLegs) {
          temp = L2Norm(signal, startOffset, modelShapes[letterIndex]);

  /* This could be made quite a bit faster since for each start offset,
   * we just add an element to the set we are minimizing over. */
          minValue = MAXFLOAT;
  /* *(costMatrix+oldLeft*numberOfChars+letterIndex-1); */
       minIndex = oldLeft;
          for (i=oldLeft;(i<oldRight)&&(i<startOffset); ++i) {
           float temp2;
           temp2 = *(costMatrix+i*numberOfChars+letterIndex-1);
           if (temp2 < minValue) {
         minIndex = i;
             minValue = temp2;
           }
          }
     *(costMatrix+startOffset*numberOfChars+letterIndex) =
          minValue + temp;
     *(pathMatrix+startOffset*numberOfChars+letterIndex) =
          minIndex;
     } /* End of if */

    } /* for startOffset */
   } /* for letterIndex */

  /* Now that all the costs have been filled in, find the cheapest */
   if (right-1+modelShapes[numberOfChars-1]->numberOfLegs+MAX_SHIFT <
     signal->numberOfLegs)
  /* In this case, the chain of letter shapes does not span the signal. */
    minValue = MAXFLOAT;
   else {
    minValue = MAXFLOAT;
    minIndex = left;
    for (i=left;(i<right)&&(i<signal->numberOfLegs); ++i) {
     float temp2;
     temp2 = *(costMatrix+i*numberOfChars+numberOfChars-1);
     if (temp2 < minValue) {
          minIndex = i;
          minValue = temp2;
     }
```

```
   }
  }

  free(costMatrix);
  free(pathMatrix);
  return minValue;
}

void PrintPath(int *pm, int width, int height, int index)
{
  int i;
  for (i=width-1;i>=0;--i) {
   printf("%d ",index);
   index = *(pm+index*width+i);
  }
  printf("\n");
}

void PrintMatrix(float *m,int width, int height)
{
  int i;
  int j;
  for (i=0;i<height;++i) {
   printf("%d: ",i);
   for (j=0;j<width;++j)
    printf("%6.2e ",*m++);
   printf("\n");
  }
}

typedef struct CTuple {
  int index;
  float value;
} CompareTuple;

int TupleLessThan(CompareTuple *t1, CompareTuple *t2)
{
  return t1->value > t2->value;
}

void L2CompareDictToString(Dictionary unknownDict,
                           char *string,
                           Dictionary modelDict,
                           BOOLEAN isBatch)
{
  CompareTuple *results;
  int i;

  if ((results = (CompareTuple *)calloc(unknownDict->numberOfEntries,
                                        sizeof(CompareTuple))) ==
     NULL)
    DoError("L2CompareDictToString: cannot allocate space.\n",NULL);

  for (i=0;i<unknownDict->numberOfEntries;++i) {
   (*(results+i)).index = i;
```

```
  (*(results+i)).value = L2CompareWithString(*(unknownDict->outlines+i),
                                             string,
                                             modelDict);
 }

 qsort(results,
       unknownDict->numberOfEntries,
       sizeof(CompareTuple),
       TupleLessThan);

 if (isBatch) {
  printf("%d(%f)\n",(*results).index,(*results).value);
 }
 else {
  printf("\n");
  for (i=0;(i<5)&&(i<unknownDict->numberOfEntries);++i)
   printf("%d: %f\n",(*(results+i)).index,(*(results+i)).value);
  printf("\n");
 }

 free(results);
}

void main(int argc,char **argv)
{
 char *unknowns,*models;
 char s[MAX_STRING_LENGTH+1];
 Dictionary unknownDict, modelDict;
 int selection;
 char *crPointer;
 BOOLEAN done = FALSE;
 BOOLEAN batch;
 char *words;

 if (argc != 3 && argc != 4) {
  printf("Usage:\n");
  printf("  %s <unknowns> <alphabet> [<batch wordlist>]\n",argv[0]);
  printf("  If the batch file is not specified, the program runs\n");
  printf("  in interactive mode.\n");
  exit(-1);
 }

 unknowns = argv[1];
 models = argv[2];
 if (argc == 4) {
  batch = TRUE;
  words = argv[3];
 } else
  batch = FALSE;

 unknownDict = ReadDictionary(unknowns);
 modelDict = ReadDictionary(models);

 if (batch) {
  FILE *fp;
```

```
  if ((fp = fopen(words,"r"))= =NULL)
   DoError("l2Norm: can't open input file %s.\n",words);
  while (!done) {
   fgets(s,MAX_STRING_LENGTH,fp);
   if ((s[0] = = '\0') || (s[0] = = '\n'))
       done = TRUE;
   else {
       crPointer = strchr(s,'\n');
       if (crPointer != NULL)
        *crPointer = '\0';
       printf("%s: ",s);
       L2CompareDictToString(unknownDict,s,modelDict,TRUE);
   }
  }
 }
 else {
  while (!done) {
   printf("Enter a word to search for: ");
   fgets(s,MAX_STRING_LENGTH,stdin);
   if ((s[0] = = '\0') || (s[0] = = '\n'))
       done = TRUE;
   else {
       crPointer = strchr(s,'\n');
       if (crPointer != NULL)
        *crPointer = '\0';
       printf("Comparing shape %s to the image\n",s);
       L2CompareDictToString(unknownDict,s,modelDict,FALSE);
   }
  }
 }
}
```

Jun 18 17:00 1991 l2Norm2.c

```
#include <stdio.h>
#include <values.h>
#include <string.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "dict.h"

#define MAX_STRING_LENGTH 30
#define MAX_SIGNAL_LENGTH 300
#define MAX_SHIFT 10

#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))

typedef struct {
 int numberOfSymbols;
 int signalLength;
 float *costs;
} *CorrelationSet,CorrelationSetBody;

CorrelationSet NewCorrelationSet(int numberOfSymbols,int signalLength)
{
 CorrelationSet temp;
 if ((temp = (CorrelationSet)calloc(1,sizeof(CorrelationSetBody)))==NULL)
  DoError("NewCorrelationSet: cannot allocate space.\n",NULL);
 temp->numberOfSymbols = numberOfSymbols;
 temp->signalLength = signalLength;
 if ((temp->costs = (float *)calloc(numberOfSymbols*signalLength,sizeof(float)))==NULL)
  DoError("NewCorrelationSet: cannot allocate space.\n",NULL);
 return temp;
}

#ifdef foo
float CorrelationValue(CorrelationSet c,int symbol,int offset)
{
 return *(c->costs+symbol*c->signalLength+offset);
}

void SetCorrelationValue(CorrelationSet c,int symbol,int offset,float value)
{
 *(c->costs+symbol*c->signalLength+offset) = value;
}
#endif
#define CorrelationValue(c,s,o) (*((c)->costs+(s)*(c)->signalLength+(o)))
#define SetCorrelationValue(c,s,o,v) (*((c)->costs+(s)*(c)->signalLength+(o))=(v))

int CorrelationSetSize(CorrelationSet c)
{
 return c->numberOfSymbols;
}
```

```
int CorrelationSetWidth(CorrelationSet c)
{
 return c->signalLength;
}

void PrintCorrelation(CorrelationSet c,int character)
{
 int i;

 for (i=0;i<c->signalLength;++i) {
  printf("%d:%6.2f\n",i,*(c->costs+character*c->signalLength+i));
 }
 printf("\n");
}

float L2Norm(OutlinePair signal, int startOffset,
         OutlinePair model)
{
 float *top1,*top2,*bottom1,*bottom2;
 int i,overlap;
 float sum;
 float temp;

 if ((startOffset < 0) ||
   (startOffset + model->numberOfLegs > signal->numberOfLegs))
  DoError("L2Norm: the model must overlap the signal.\n",NULL);

 top1 = signal->top+startOffset;
 top2 = model->top;
 bottom1 = signal->bottom+startOffset;
 bottom2 = model->bottom;

 overlap = signal->numberOfLegs - startOffset;
 if (overlap > model->numberOfLegs)
  overlap = model->numberOfLegs;

 for (i=0,sum=0;i<overlap;++i) {
  temp = *top1++ - *top2++;
  sum += temp * temp;
  temp = *bottom1++ - *bottom2++;
  sum += temp * temp;
 }

 return sum;
}

CorrelationSet PreProcessSignalWithChars(OutlinePair signal,Dictionary charDict)
{
 CorrelationSet cSet;
 int thisChar,offset;
 OutlinePair charSignal;

 cSet = NewCorrelationSet(charDict->numberOfEntries,signal->numberOfLegs);

 for (thisChar = 0; thisChar < charDict->numberOfEntries; ++thisChar) {
```

```
  charSignal = *(charDict->outlines+thisChar);
  for (offset = 0; offset < signal->numberOfLegs-charSignal->numberOfLegs+1;
 ++offset)
   SetCorrelationValue(cSet,thisChar,offset,L2Norm(signal,offset,charSignal));
 }
 return cSet;
}

CorrelationSet *PreProcessDictionaryWithChars(Dictionary signalDict,Dictionary charDict)
{
 CorrelationSet *correlationSets;
 int thisWord;

 correlationSets = (CorrelationSet
*)calloc(signalDict->numberOfEntries,sizeof(CorrelationSet));
 if (correlationSets == NULL)
  DoError("PreProcessDicitonary: cannot allocate space.\n",NULL);
 for (thisWord = 0; thisWord < signalDict->numberOfEntries; ++thisWord) {
  *(correlationSets+thisWord) =
PreProcessSignalWithChars(*(signalDict->outlines+thisWord),charDict);
  printf("%d ",thisWord);
 }
 return correlationSets;
}

CorrelationSet PreProcessSignalWithBlanks(OutlinePair signal)
{
 CorrelationSet cSet;
 int blankWidth,offset;
 int numberOfLegs = signal->numberOfLegs;

 cSet = NewCorrelationSet(MAX_SHIFT,numberOfLegs);

 for (offset = 0; offset < numberOfLegs; ++offset) {
  SetCorrelationValue(cSet,0,offset,0);
 }
 for (offset = 0; offset < numberOfLegs; ++offset) {
  float top,bottom;
  top = *(signal->top+offset);
  bottom = *(signal->bottom+offset);
  SetCorrelationValue(cSet,1,offset,top*top+bottom*bottom);
 }
 for (blankWidth = 2; blankWidth < MAX_SHIFT; ++blankWidth) {
  for (offset = 0; offset < numberOfLegs-blankWidth+1; ++offset) {
   float top,bottom,temp;
   top = *(signal->top+offset+blankWidth-1);
   bottom = *(signal->bottom+offset+blankWidth-1);
   temp = top*top+bottom*bottom+CorrelationValue(cSet,blankWidth-1,offset);
   SetCorrelationValue(cSet,blankWidth,offset,temp);
  }
 }
 return cSet;
}

CorrelationSet *PreProcessDictionaryWithBlanks(Dictionary signalDict)
```

```
{
 CorrelationSet *correlations;
 int thisWord;

 correlations = (CorrelationSet
*)calloc(signalDict->numberOfEntries,sizeof(CorrelationSet));

 for (thisWord = 0; thisWord < signalDict->numberOfEntries; ++thisWord) {
  *(correlations+thisWord) =
PreProcessSignalWithBlanks(*(signalDict->outlines+thisWord));
  printf("%d ",thisWord);
 }
 return correlations;
}

int LookupShapeIndex(char c, Dictionary models)
{
/* dictionary file has the following order:
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 0123456789
 ,.
*/
 int shapeIndex;
 if ((c>='a'&&c<='z'))
  shapeIndex = c-'a';
 else if (c == ',')
  shapeIndex = 26;
 else if (c == '.')
  shapeIndex = 27;
 else
  DoError("LookupShape: have no shape one of the characters.\n",NULL);
 return shapeIndex;
}


float L2CompareWithString(int signalIndex,
                          char *string,
                          CorrelationSet charCorrelations,
                          CorrelationSet blankCorrelations,
                          Dictionary signalDict,
                          Dictionary models)
{
/* Allocate space for dynamic programming array. */
/* For now, be a space hog. */
 float costMatrix[MAX_SIGNAL_LENGTH][MAX_STRING_LENGTH];
 int pathMatrix[MAX_SIGNAL_LENGTH][MAX_STRING_LENGTH];
 char *charCursor;
 OutlinePair modelShapes[MAX_STRING_LENGTH];
 int modelIndices[MAX_STRING_LENGTH];
 int numberOfChars;

 int letterIndex, startOffset;
 int left,right;
 int searchLeft,searchRight,rightEdge;
```

```
 float minValue;
 float temp;
 int i,oldLeft,oldRight,minIndex;
 int signalLength;

 signalLength = (*(signalDict->outlines+signalIndex))->numberOfLegs;

/* Make sure input string is not too long. */
 numberOfChars = strlen(string);
 if (numberOfChars >= MAX_STRING_LENGTH)
  DoError("L2CompareWithString: string is too long.\n",NULL);

/* Make sure signal is not too long. */
 if (signalLength >= MAX_SIGNAL_LENGTH)
  DoError("L2CompareWithString: signal is too long.\n",NULL);

/* Lookup the indices of the signals corresponding to the characters in the string. */
 charCursor = string;
 for (i=0;i<numberOfChars;++i) {
  modelIndices[i] = LookupShapeIndex(*charCursor++,models);
  modelShapes[i] = *(models->outlines+modelIndices[i]);
 }

/* Since the cost matrix is larger than the region containing valid
 * alignments, first fill in the array with large costs. Later, some
 * of these will be overwritten. */
/* WARNING: does MAXFLOAT + smallFloat == MAXFLOAT or does it roll? */
 {
  float *cursor;
  cursor = &(costMatrix[0][0]);
  for (i=0;i<MAX_SIGNAL_LENGTH*MAX_STRING_LENGTH;++i)
   *cursor++ = MAXFLOAT;
 }

/* Fill in leftmost column */
 left = 0;
 right = MIN(MAX_SHIFT,signalLength-modelShapes[0]->numberOfLegs);
 for (startOffset=left;startOffset<right;++startOffset)
  costMatrix[startOffset][0] = CorrelationValue(blankCorrelations,startOffset,startOffset)
+
                CorrelationValue(charCorrelations,modelIndices[0],startOffset);

/* Now do the rest of the columns */
 for (letterIndex = 1; letterIndex < numberOfChars; ++letterIndex) {
  oldLeft = left;
  oldRight = right;
  left += modelShapes[letterIndex-1]->numberOfLegs;
/* If string of characters is too long for this signal, abort by returning a large cost. */
  if (left>=signalLength)
   return MAXFLOAT;
  right += modelShapes[letterIndex-1]->numberOfLegs + MAX_SHIFT;
  right = MIN(right,signalLength-modelShapes[letterIndex]->numberOfLegs+1);

  for (startOffset=left;startOffset<right;++startOffset) {
   temp = CorrelationValue(charCorrelations,modelIndices[letterIndex],startOffset);
```

```
/* This could be made quite a bit faster since for each start offset,
 * we just add an element to the set we are minimizing over. */

    searchLeft = startOffset-modelShapes[letterIndex-1]->numberOfLegs-MAX_SHIFT;
    searchLeft = MAX(searchLeft,oldLeft);
    rightEdge = searchLeft+modelShapes[letterIndex-1]->numberOfLegs;
    searchRight = startOffset-modelShapes[letterIndex-1]->numberOfLegs;
    searchRight = MIN(searchRight,oldRight);

    minIndex = searchLeft;
    minValue = costMatrix[searchLeft][letterIndex-1] +
          CorrelationValue(blankCorrelations,startOffset-rightEdge,rightEdge);

    for (i=searchLeft; i<searchRight; ++i,++rightEdge) {
       float temp;
       temp = costMatrix[i][letterIndex-1] +
        CorrelationValue(blankCorrelations,startOffset-rightEdge,rightEdge);
       if (temp < minValue) {
        minIndex = i;
        minValue = temp;
       }
    }

    costMatrix[startOffset][letterIndex] = minValue+temp;
    pathMatrix[startOffset][letterIndex] = minIndex;
   } /* for startOffset */
  } /* for letterIndex */


 /* fill in the costs for blanks at the end of the word */
  rightEdge = left+modelShapes[letterIndex-1]->numberOfLegs;
  for (startOffset=left;startOffset<right;++startOffset,++rightEdge) {
   if (rightEdge+MAX_SHIFT >= signalLength) {
    costMatrix[startOffset][letterIndex-1] +=
        CorrelationValue(blankCorrelations,signalLength-1-rightEdge,rightEdge);
   }
   else {
    /* this chain of letters does not span the word */
    costMatrix[startOffset][letterIndex-1] = MAXFLOAT;
   }
  }

 /* keep minIndex for debugging pruposes */
  minIndex = left;
  minValue = costMatrix[left][letterIndex-1];
  for (i=left; i<right; ++i) {
   float temp;
   temp = costMatrix[i][letterIndex-1];
   if (temp < minValue) {
    minIndex = i;
    minValue = temp;
   }
  }
```

```
 return minValue;
}

void PrintPath(int *pm, int width, int height, int index)
{
 int i;
 for (i=width-1;i>=0;--i) {
  printf("%d ",index);
  index = *(pm+index*width+i);
 }
 printf("\n");
}

void PrintMatrix(float *m,int width, int height)
{
 int i;
 int j;
 for (i=0;i<height;++i) {
  printf("%d: ",i);
  for (j=0;j<width;++j)
   printf("%6.2e ",*m++);
  printf("\n");
 }
}

typedef struct CTuple {
 int index;
 float value;
} CompareTuple;

int TupleLessThan(CompareTuple *t1, CompareTuple *t2)
{
 return t1->value > t2->value;
}

void L2CompareDictToString(char *string,
                    CorrelationSet *charCorrelations,
                    CorrelationSet *blankCorrelations,
                    Dictionary signalDict,
                    Dictionary modelDict,
                    BOOLEAN isBatch)
{
 CompareTuple *results;
 int i;

 if ((results = (CompareTuple *)calloc(signalDict->numberOfEntries,
                              sizeof(CompareTuple))) ==
   NULL)
  DoError("L2CompareDictToString: cannot allocate space.\n",NULL);

 for (i=0;i<signalDict->numberOfEntries;++i) {
  (*(results+i)).index = i;
  (*(results+i)).value = L2CompareWithString(i,string,
                              *(charCorrelations+i),
                              *(blankCorrelations+i),
```

```
                                        signalDict,
                                        modelDict);
  }

  qsort(results,
        signalDict->numberOfEntries,
        sizeof(CompareTuple),
        TupleLessThan);

  if (isBatch) {
   printf("%d(%f)\n",(*results).index,(*results).value);
  }
  else {
   printf("\n");
   for (i=0;(i<5)&&(i<signalDict->numberOfEntries);++i)
    printf("%d: %f\n",(*(results+i)).index,(*(results+i)).value);
   printf("\n");
  }

  free(results);
}

void PrintDictStats(Dictionary dict)
{
 int i,sum=0;
 printf("Dictionary has %d entries.\n",dict->numberOfEntries);
 for (i=0;i<dict->numberOfEntries;++i)
  sum += (*(dict->outlines+i))->numberOfLegs;
 printf("The total length of the shape contours is %d pixels.\n",sum);
}

void main(int argc,char **argv)
{
 char *unknowns,*models;
 char s[MAX_STRING_LENGTH+1];
 Dictionary unknownDict, modelDict;
 int selection;
 char *crPointer;
 BOOLEAN done = FALSE;
 BOOLEAN batch;
 char *words;
 CorrelationSet *charCorrelations;
 CorrelationSet *blankCorrelations;

 if (argc != 3 && argc != 4) {
  printf("Usage:\n");
  printf("  %s <unknowns> <alphabet> [<batch wordlist>]\n",argv[0]);
  printf("  If the batch file is not specified, the program runs\n");
  printf("  in interactive mode.\n");
  exit(-1);
 }

 unknowns = argv[1];
 models = argv[2];
 if (argc == 4) {
```

```
      batch = TRUE;
      words = argv[3];
    } else
      batch = FALSE;

    unknownDict = ReadDictionary(unknowns);
    modelDict = ReadDictionary(models);

    PrintDictStats(unknownDict);
    printf("Preprocessing . . .\n");
    charCorrelations = PreProcessDictionaryWithChars(unknownDict,modelDict);
    blankCorrelations = PreProcessDictionaryWithBlanks(unknownDict);
    printf("done.\n");

    if (batch) {
     FILE *fp;
     if ((fp = fopen(words,"r"))= =NULL)
      DoError("l2Norm: can't open input file %s.\n",words);
     while (!done) {
      fgets(s,MAX_STRING_LENGTH,fp);
      if ((s[0] = = '\0') || (s[0] = = '\n'))
          done = TRUE;
      else {
          crPointer = strchr(s,'\n');
          if (crPointer != NULL)
            *crPointer = '\0';
          printf("%s: ",s);
          L2CompareDictToString(s,charCorrelations,blankCorrelations,unknownDict,modelDict,TRUE);
      }
     }
    }
    else {
     while (!done) {
      printf("Enter a word to search for: ");
      fgets(s,MAX_STRING_LENGTH,stdin);
      if ((s[0] = = '\0') || (s[0] = = '\n'))
          done = TRUE;
      else {
          crPointer = strchr(s,'\n');
          if (crPointer != NULL)
            *crPointer = '\0';
          printf("Comparing shape %s to the image\n",s);
          L2CompareDictToString(s,charCorrelations,blankCorrelations,unknownDict,modelDict,FALSE);
      }
     }
    }
   }
```

Jan 15 21:32 1991 match.c

```
/*
 * match.c
 * align 2 sequences
 *
 * run as: match seq1 seq2
 *
 */

/*
 * TO DO: 1) don't compute over parts of array outside of constraints
 *      2) distance score for top and bottom paths
 */

#include <stdio.h>
#include <math.h>

#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"
#include "diff.h"
#include "diff2.h"
#include "match.h"

#ifndef MAXLINE
#define MAXLINE 256
#endif

#ifndef MAXNAME
#define MAXNAME 128
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

int matchcntr = 1; /* used for writing out set number of matches */

/*
void
main(argc,argv)
int argc;
char *argv[];
{
*/
 /*
  * read in multiple parameter files, write out selected fields as shorts
```

```
  */
 /*
  int i,j;
  int seqlength;
  float matchvecs();

  float test[MAXSEQLENGTH];
  float ref[MAXSEQLENGTH];
 */
  /*
   * read in args
   */
 /*
  debug = FALSE;

  for (;argc > 1 && (argv[1][0] == '-'); argc--, argv++)
   {
    switch (argv[1][1])
        {
        case 'd':
         debug = TRUE;
         break;
        case 'h':
         horweight = (float)atoi(&argv[1][2]);
         break;
        case 'v':
         verweight = (float)atoi(&argv[1][2]);
         break;
        case 'x':
         diagweight = (float)atoi(&argv[1][2]);
         break;
        default:
         printf ("match: unknown switch %s.\n", argv[1]);
         exit (1);
        }
   }

  if (argc != 1)
   {
    printf ("Usage: match [-b<begsamp> -d(debug) -e<endsamp>\n");
    printf ("argc: %d\n", argc);
    exit (1);
   }
 */
  /* debugging */
 /* for (i = 0; i < 5; i++)
    test[i] = (float)i;
  for (i = 5; i < 10; i++)
    test[i] = (float)(.5 * (i - 4) + 5);
  for (i = 0; i < 5; i++)
    ref[i] = 1.5 * i;
  matchvecs(test, 10, ref, 5);

 }
 */
```

```
/*
float DPDiffPair(OutlinePair one, OutlinePair two)
{
 if (one == two){
  printf("matches\n");
  return(0.0);
 }
 else {
  printf("no match\n");
  return(1.0);
 }
}
*/

float DPDiffPair(OutlinePair one, OutlinePair two)
 /*
  * question, should top and bottom distance be forced to be computed together?
  * use another distance score to check how far off the two are?
  */

{
 float topscore;
 float bottomscore;

 if (debug) printf("top: ");
 topscore = matchvecs(one->top, one->numberOfLegs,
                 two->top, two->numberOfLegs);
 if (debug) printf(" bottom: ");
 bottomscore = matchvecs(one->bottom, one->numberOfLegs,
                    two->bottom, two->numberOfLegs);
 return (topscore + bottomscore);
}

float matchvecs(float *Vec1, int lenVec1, float *Vec2, int lenVec2)
   /*
    * Computes the best path between one and two.
    * Allows 2/1 expansion/compression
    */
{
 float dist, mindist, hor, vert, diag;
 float bestscore;
 int i1, i2;
 int xdir, ydir;

 elt *array[MAXSEQLENGTH][MAXSEQLENGTH];
 elt *aelt;

 /* initialize array */

 for (i1 = 0; i1 < lenVec1; i1++) {
  for (i2 = 0; i2 < lenVec2; i2++) {
   array[i1][i2] = (elt *) malloc(sizeof (elt));
   if (array[i1][i2] == NULL) {
      fprintf(stderr,
```

```
                "Sorry, not enough space to malloc array elts in pl_matchvecs\n BYE!");
            exit(1);
        }
      }
    }

    /*
     * compute match
     */
    /* initialize */
    aelt = array[0][0];
    aelt->cost = sq_distance(Vec1[0], Vec2[0]);
    aelt->xptr = 0;
    aelt->yptr = 0;
    /* bottom row */
    i2 = 0;
    for (i1 = 1; i1 < lenVec1; i1++) {
      dist = sq_distance(Vec1[i1], Vec2[i2]);
      aelt = array[i1][i2];
      aelt->cost = array[i1 - 1][i2]->cost + horweight * dist;
      aelt->xptr = -1;
      aelt->yptr = 0;
    }
    /* left column */
    i1 = 0;
    for (i2 = 1; i2 < lenVec2; i2++) {
      dist = sq_distance(Vec1[i1], Vec2[i2]);
      aelt = array[i1][i2];
      aelt->cost = array[i1][i2 - 1]->cost + verweight * dist;
      aelt->xptr = 0;
      aelt->yptr = -1;
    }
    /* middle */
    for (i1 = 1; i1 < lenVec1; i1++) {
      for (i2 = 1; i2 < lenVec2; i2++) {
        dist = sq_distance(Vec1[i1], Vec2[i2]);
        hor = array[i1 - 1][i2]->cost + horweight * dist;
        xdir = -1;
        ydir = 0;
        mindist = hor;
        vert = array[i1][i2 - 1]->cost + verweight * dist;
        if (vert < mindist) {
            xdir = 0;
            ydir = -1;
            mindist = vert;
        }
        diag = array[i1 - 1][i2 - 1]->cost + diagweight * dist;
        if (diag < mindist) {
            xdir = -1;
            ydir = -1;
            mindist = diag;
        }
        aelt = array[i1][i2];
        aelt-> cost = mindist;
        aelt-> xptr = xdir;
```

```
    aelt->yptr = ydir;
   }
  }

  bestscore = best_score(array, lenVec1, lenVec2);
#ifdef foo
  if (debug) {
   print_best_path(array, lenVec1, lenVec2);
/*   print_array_costs(array, lenVec1, lenVec2);
   print_array_dirs(array, lenVec1, lenVec2);
*/
  printf("best score: %f\n", bestscore);
  }
#endif

  for (i1 = 0; i1 < lenVec1; i1++) {
   for (i2 = 0; i2 < lenVec2; i2++) {
    free(array[i1][i2]);
   }
  }

  return(bestscore);
}

float
sq_distance(float x1, float x2)
{
  float dist;
  float epsilon = .001;

  /*
   * quantization makes many values identical, use of epsilon encourages shortest path
   */

  dist = x1 - x2;
  dist *= dist;
  dist += epsilon;
  return(dist);
}

/*
float parallel_distance(OutlinePair one, OutlinePair two, int ptr1, int ptr2)
{
  float topdist, bottomdist;

  topdist = one->top[ptr1] - two->top[ptr2];
  topdist *= topdist;

  bottomdist = one->bottom[ptr1] - two->bottom[ptr2];
  bottomdist *= bottomdist;

  return(topdist + bottomdist);
}
*/
```

```
float
best_score (elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2)
{
 /* assume all of Vec1 and Vec2 are used, so just take value at end */

 return(array[lenVec1 - 1][lenVec2 - 1]->cost);
}

/*
 * **********************
 * debugging functions
 */
#ifdef foo
void
print_best_path(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2)
{
 char path[MAXNAME];
 int x, y;
 elt *aelt;
 FILE *ofp;

 x = lenVec1 - 1;
 y = lenVec2 - 1;

 sprintf(path, "/net/piglet/piglet/speech/fchen/pics/paths/p%d.txt", FileCountY);

 ofp = fopen (path, "a");
 if(ofp == NULL)
  printf("Cannot open output file %s.\n", path);

/* fprintf(ofp, " %3s %3s %6s\n", "x", "y", "cost");
 */
 while (x > 0 || y > 0) {
  aelt = array[x][y];
  fprintf(ofp, " %3d %3d %6.2f\n", x, y, aelt->cost);
  x += aelt->xptr;
  y += aelt->yptr;
 }
/* fprintf(ofp, " \"match %d\n\n", matchcntr++);
*/
 fprintf(ofp, " \"match %d %d\n\n", FileCountX, FileCountY);
 fclose(ofp);
}
#endif
static float sqr(float x)
{
 return x*x;
}

void print_best_path(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2,
              char *outFileName)
{
 int x, y;
 elt *aelt;
 FILE *outFile;
```

```
  float dist = 0;

  x = lenVec1 - 1;
  y = lenVec2 - 1;

  if ((outFile = (FILE *)fopen(outFileName,"w"))==NULL)
   DoError("single: Cannot open output file %s.\n", outFileName);

  while (x > 0 || y > 0) {
   aelt = array[x][y];
   fprintf(outFile, " %3d %3d %6.2f\n", x, y, aelt->cost);
   dist += sqrt(sqr(aelt->xptr)+sqr(aelt->yptr));
   x += aelt->xptr;
   y += aelt->yptr;
  }
  fclose(outFile);
  printf("distance = %f\n",dist);
}


void
print_array_costs(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2)
{

  int x, y;

  for (y = 0; y < lenVec2; y++){
   for (x = 0; x < lenVec1; x++){
    printf("%7.2f ", array[x][y]->cost);
   }
   printf("\n");
  }
}

void
print_array_dirs(elt *array[][MAXSEQLENGTH], int lenVec1, int lenVec2)
{

  int x, y;

  for (y = 0; y < lenVec2; y++){
   for (x = 0; x < lenVec1; x++){
    printf("%2d:%2d  ", array[x][y]->xptr, array[x][y]->yptr);
   }
   printf("\n");
  }
}
```

Jul 7 14:28 1991 matchparallel.c

```
/*
 * matchparallel.c
 * align 2 sequences
 *
 * dependent on match.c
 */

/*
 * TO DO: 1) don't compute over parts of array outside of constraints
 *
 */

#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"
#include "diff.h"
#include "diff2.h"
#include "match.h"
#include "matchparallel.h"

#ifndef MAXLINE
#define MAXLINE 256
#endif

#ifndef MAXNAME
#define MAXNAME 128
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef max
#define max(a,b) ((a) > (b) ? (a) : (b))
#endif

#ifndef min
#define min(a,b) ((a) < (b) ? (a) : (b))
#endif

/*
 *********************
 * parallel match with full search
 *********************
```

```
*/

float pl_DPDiffPair(OutlinePair one, OutlinePair two, char *pathFile)
 /*
  * question, should top and bottom distance be forced to be computed together?
  * use another distance score to check how far off the two are?
  */

{
 float score;

 score = pl_matchvecs(one->top, one->bottom, one->numberOfLegs,
                     two->top, two->bottom, two->numberOfLegs,
                       pathFile);
 return (score);
}

float pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1,
                   float *Vec2t, float *Vec2b, int lenVec2,
                   char *pathFile)
  /*
   * Computes the best path between one and two.
   * Allows 2/1 expansion/compression
   */
{
 float dist, mindist, hor, vert, diag;
 float bestscore;
 int i1, i2;
 int xdir, ydir;

 elt *array[MAXSEQLENGTH][MAXSEQLENGTH];
 elt *aelt;

 /* initialize array */

 for (i1 = 0; i1 < lenVec1; i1++) {
  for (i2 = 0; i2 < lenVec2; i2++) {
   array[i1][i2] = (elt *) malloc(sizeof (elt));
   if (array[i1][i2] == NULL) {
       fprintf(stderr,
               "Sorry, not enough space to malloc array elts in pl_matchvecs\n BYE!");
       exit(1);
   }
  }
 }

 /*
  * compute match
  */
 /* initialize */
 aelt = array[0][0];
 aelt->cost = sq_distance(Vec1t[0], Vec2t[0]) + sq_distance(Vec1b[0], Vec2b[0]);
 aelt->xptr = 0;
 aelt->yptr = 0;
 /* bottom row */
```

```
*/

float pl_DPDiffPair(OutlinePair one, OutlinePair two, char *pathFile)
 /*
  * question, should top and bottom distance be forced to be computed together?
  * use another distance score to check how far off the two are?
  */

{
 float score;

 score = pl_matchvecs(one->top, one->bottom, one->numberOfLegs,
                 two->top, two->bottom, two->numberOfLegs,
                   pathFile);
 return (score);
}

float pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1,
                float *Vec2t, float *Vec2b, int lenVec2,
                char *pathFile)
  /*
   * Computes the best path between one and two.
   * Allows 2/1 expansion/compression
   */
{
 float dist, mindist, hor, vert, diag;
 float bestscore;
 int i1, i2;
 int xdir, ydir;

 elt *array[MAXSEQLENGTH][MAXSEQLENGTH];
 elt *aelt;

 /* initialize array */

 for (i1 = 0; i1 < lenVec1; i1++) {
  for (i2 = 0; i2 < lenVec2; i2++) {
   array[i1][i2] = (elt *) malloc(sizeof (elt));
   if (array[i1][i2] == NULL) {
       fprintf(stderr,
             "Sorry, not enough space to malloc array elts in pl_matchvecs\n BYE!");
       exit(1);
   }
  }
 }

 /*
  * compute match
  */
 /* initialize */
 aelt = array[0][0];
 aelt->cost = sq_distance(Vec1t[0], Vec2t[0]) + sq_distance(Vec1b[0], Vec2b[0]);
 aelt->xptr = 0;
 aelt->yptr = 0;
 /* bottom row */
```

```
i2 = 0;
for (i1 = 1; i1 < lenVec1; i1++) {
 dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
 aelt = array[i1][i2];
 aelt->cost = array[i1 - 1][i2]->cost + horweight * dist;
 aelt->xptr = -1;
 aelt->yptr = 0;
}
/* left column */
i1 = 0;
for (i2 = 1; i2 < lenVec2; i2++) {
 dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
 aelt = array[i1][i2];
 aelt->cost = array[i1][i2 - 1]->cost + verweight * dist;
 aelt->xptr = 0;
 aelt->yptr = -1;
}
/* middle */
for (i1 = 1; i1 < lenVec1; i1++) {
 for (i2 = 1; i2 < lenVec2; i2++) {
  dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
  hor = array[i1 - 1][i2]->cost + horweight * dist;
  xdir = -1;
  ydir = 0;
  mindist = hor;
  vert = array[i1][i2 - 1]->cost + verweight * dist;
  if (vert < mindist) {
      xdir = 0;
      ydir = -1;
      mindist = vert;
  }
  diag = array[i1 - 1][i2 - 1]->cost + diagweight * dist;
  if (diag < mindist) {
      xdir = -1;
      ydir = -1;
      mindist = diag;
  }
  aelt = array[i1][i2];
  aelt-> cost = mindist;
  aelt-> xptr = xdir;
  aelt-> yptr = ydir;
 }
}

bestscore = best_score(array, lenVec1, lenVec2);
if (pathFile)
 print_best_path(array, lenVec1, lenVec2,pathFile);


for (i1 = 0; i1 < lenVec1; i1++) {
 for (i2 = 0; i2 < lenVec2; i2++) {
  free(array[i1][i2]);
 }
}
```

```
 return(bestscore);
}


/*
 * ********************
 * faster parallel match
 * not optimal because warp is limited to swath of width "bw"
 * ********************
 */

float faster_pl_DPDiffPair(OutlinePair one, OutlinePair two, char *pathFile)
 /*
  * question, should top and bottom distance be forced to be computed together?
  * use another distance score to check how far off the two are?
  */

{
 float score;

 score = faster_pl_matchvecs(one->top, one->bottom, one->numberOfLegs,
                             two->top, two->bottom, two->numberOfLegs,
                              pathFile);
 return (score);
}

float faster_pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1,
                          float *Vec2t, float *Vec2b, int lenVec2,
                          char *pathFile)
   /*
    * Computes the best path between one and two.
    * Allows 2/1 expansion/compression only within a band
    */
{
 float dist, mindist, hor, vert, diag;
 float bestscore;
 float ratio;
 int i1, i2;
 int xdir, ydir;
 int beg, end, center;
 int b;              /* pointer to border */
 int border;                 /* width of border on right side of swath */

 elt *array[MAXSEQLENGTH][MAXSEQLENGTH];
 elt *aelt;

 float infinity = 1.0e30;
 int bw = 20;

 ratio = (float)lenVec1/ (float)lenVec2;
 border = (int) (ratio + .999999);
/* if (debug)
  printf("ratio: %f\n", ratio);
*/ /* initialize array */
```

```
    for (i1 = 0; I1 < lenVec1; i1++) {
     for (i2 = 0; i2 < lenVec2; i2++) {
      array[i1][i2] = (elt *) malloc(sizeof (elt));
      if (array[i1][i2] == NULL) {
          fprintf(stderr,
                  "Sorry, not enough space to malloc array elts in pl_matchvecs\n BYE!");
          exit(1);
      }
     }
    }

    /*
     * compute match
     */
    /* initialize */
    aelt = array[0][0];
    aelt->cost = sq_distance(Vec1t[0], Vec2t[0]) + sq_distance(Vec1b[0], Vec2b[0]);
    aelt->xptr = 0;
    aelt->yptr = 0;
    /* bottom row */
    i2 = 0;
    end = bw + border + 1;
    for (i1 = 1; i1 < end; i1++) {
     dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
     aelt = array[i1][i2];
     aelt->cost = array[i1 - 1][i2]->cost + horweight * dist;
     aelt->xptr = -1;
     aelt->yptr = 0;
    }
    /*
     * swath
     *
     * set the elt before beg and at end to infinity, then the compute distances normally
     * for the row
     */
    for (i2 = 1; i2 < lenVec2; i2++) {
     center = i2 * ratio;
     beg = max(1, center - bw);
     end = min(lenVec1, center + bw + 1);
   /*  if (debug)
      printf("center: %d, beg: %d, end: %d\n", center, beg, end);
   */  /* beg */
     aelt = array[beg - 1][i2];
     aelt->xptr = 0;
     aelt->yptr = -1;
     if (beg == 1) {
      dist = sq_distance(Vec1t[0], Vec2t[i2]) + sq_distance(Vec1b[0], Vec2b[i2]);
      aelt->cost = array[0][i2 - 1]->cost + verweight * dist;
     }
     else {
      aelt->cost = infinity;
     }
     /* end */
   /*  if (end < lenVec1) {
   */
```

```
    for (b = end; b < min(end+border,lenVec1); b++) {
/*    if (debug)
        printf("b: %d ", b);
*/    aelt = array[b][i2];
      aelt->cost = infinity;
      aelt->xptr = -1;
      aelt->yptr = 0;
    }
    for (i1 = beg; i1 < end; i1++) {
      dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
      hor = array[i1 - 1][i2]->cost + horweight * dist;
      xdir = -1;
      ydir = 0;
      mindist = hor;
      vert = array[i1][i2 - 1]->cost + verweight * dist;
      if (vert < mindist) {
          xdir = 0;
          ydir = -1;
          mindist = vert;
      }
      diag = array[i1 - 1][i2 - 1]->cost + diagweight * dist;
      if (diag < mindist) {
          xdir = -1;
          ydir = -1;
          mindist = diag;
      }
      aelt = array[i1][i2];
      aelt -> cost = mindist;
      aelt -> xptr = xdir;
      aelt -> yptr = ydir;
    }
  }

  bestscore = best_score(array, lenVec1, lenVec2);
  if (pathFile)
   print_best_path(array, lenVec1, lenVec2,pathFile);

  for (i1 = 0; i1 < lenVec1; i1++) {
   for (i2 = 0; i2 < lenVec2; i2++) {
    free(array[i1][i2]);
   }
  }

  return(bestscore);
}


/*
 * ********************
 * fastest parallel match
 * warp limited to swath bw, plus no backtracking
 * ********************
 */

float simple_pl_DPDiffPair(OutlinePair one, OutlinePair two)
```

```
/*
 * question, should top and bottom distance be forced to be computed together?
 * use another distance score to check how far off the two are?
 */

{
 float score;

 score = simple_pl_matchvecs(one->top, one->bottom, one->numberOfLegs,
                     two->top, two->bottom, two->numberOfLegs);
 return (score);
}

void PrintArrayRow(float *array,int width)
{
 int i;
 for (i=0;i<width;++i)
  printf("%2.2f ",*array++);
 printf("\n");
}

void PrintArray(float *array,int height,int width,int signalWidth)
{
 int i;
 for (i=0;i<height;++i) {
  printf("%d: ",i);
  PrintArrayRow(array+i*width,signalWidth);
 }
}

float simple_pl_matchvecs(float *Vec1t, float *Vec1b, int lenVec1, float *Vec2t, float
*Vec2b, int lenVec2)
  /*
   * Computes the best path between one and two within a band.
   * Allows 2/1 expansion/compression only within a band.
   */
{
 float dist, mindist, hor, vert, diag;
 float bestscore;
 float ratio;
 int i1, i2;
 int xdir, ydir;
 int beg, end, center;
 int b;              /* pointer to border */
 int border;              /* width of border on right side of swath */

 float array[MAXSEQLENGTH][MAXSEQLENGTH];

 float infinity = 1.0e30;
 int bw = 20;

 ratio = (float)lenVec1/ (float)lenVec2;
 border = (int) (ratio + .999999);
/* if (debug)
  printf("ratio: %f\n", ratio);
```

```
*/ /* initialize array */


 /*
  * compute match
  */
 /* initialize */
 array[0][0] = sq_distance(Vec1t[0], Vec2t[0]) + sq_distance(Vec1b[0], Vec2b[0]);

 /* bottom row */
 i2 = 0;
 end = bw + border + 1;
 for (i1 = 1; i1 < end; i1++) {
  dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
  array[i1][i2] = array[i1 - 1][i2] + horweight * dist;
 }
 /*
  * swath
  *
  * set the elt before beg and at end to infinity, then the compute distances normally
  * for the row
  */
 for (i2 = 1; i2 < lenVec2; i2++) {
  center = i2 * ratio;
  beg = max(1, center - bw);
  end = min(lenVec1, center + bw + 1);
/*   if (debug)
    printf("center: %d, beg: %d, end: %d\n", center, beg, end);
*/  /* beg */
  if (beg == 1) {
   dist = sq_distance(Vec1t[0], Vec2t[i2]) + sq_distance(Vec1b[0], Vec2b[i2]);
   array[beg - 1][i2] = array[0][i2 - 1] + verweight * dist;
  }
  else {
   array[beg - 1][i2] = infinity;
  }
  /* end */
  for (b = end; b < min(end+border,lenVec1); b++) {
/*    if (debug)
       printf("b: %d ", b);
*/
   array[b][i2] = infinity;
  }
  for (i1 = beg; i1 < end; i1++) {
   dist = sq_distance(Vec1t[i1], Vec2t[i2]) + sq_distance(Vec1b[i1], Vec2b[i2]);
   hor = array[i1 - 1][i2] + horweight * dist;
   mindist = hor;
   vert = array[i1][i2 - 1] + verweight * dist;
   if (vert < mindist) {
       mindist = vert;
   }
   diag = array[i1 - 1][i2 - 1] + diagweight * dist;
   if (diag < mindist) {
       mindist = diag;
   }
```

```
      array[i1][i2] = mindist;
    }
  }

  bestscore = array[lenVec1 - 1][lenVec2 - 1];
  if (debug) {
    printf("best score: %f\n", bestscore);
  }

  return(bestscore);
}
```

Jul 24 17:16 1991 newL2.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "dict.h"

#define NORMAL_LENGTH (100.0)
#define MAX_SLOPE (2.0)
#define BIG_NUM (10.0e20)

void ResampleOutlinePair(OutlinePair a,float newToOldFactor)
/* Resample an outline pair using linear interpolation. */
{
 int newWidth,oldWidth,i;
 int oldLeft,oldRight;
 float oldCenter;
 float *newX,*newTop,*newBottom;
 float *xCursor,*topCursor,*bottomCursor;

 oldWidth = a->numberOfLegs;
 newWidth = irint(newToOldFactor*oldWidth);

 newX = (float *)calloc(newWidth,sizeof(float));
 newTop = (float *)calloc(newWidth,sizeof(float));
 newBottom = (float *)calloc(newWidth,sizeof(float));
 if ((newX==NULL)||(newTop==NULL)||(newBottom==NULL))
  DoError("ResampleOutlinePair: cannot allocate space.\n",NULL);

 xCursor = newX;
 topCursor = newTop;
 bottomCursor = newBottom;

 for (i=0;i<newWidth;++i) {
  oldCenter = i/(float)newWidth*(float)oldWidth;
  oldLeft = irint(floor(oldCenter));
  oldRight = irint(ceil(oldCenter));
  if (oldLeft==oldRight) {
   *xCursor++ = *(a->x+oldLeft);
   *topCursor++ = *(a->top+oldLeft);
   *bottomCursor++ = *(a->bottom+oldLeft);
  }
  else {
   float slope;
   slope = *(a->x+oldRight)-*(a->x+oldLeft);
   *xCursor++ = *(a->x+oldLeft) + (oldCenter-oldLeft)*slope;
   slope = *(a->top+oldRight)-*(a->top+oldLeft);
   *topCursor++ = *(a->top+oldLeft) + (oldCenter-oldLeft)*slope;
   slope = *(a->bottom+oldRight)-*(a->bottom+oldLeft);
   *bottomCursor++ = *(a->bottom+oldLeft) + (oldCenter-oldLeft)*slope;
  }
 }
```

```
  free(a->x);
  free(a->top);
  free(a->bottom);

  a->x = newX;
  a->top = newTop;
  a->bottom = newBottom;
  a->numberOfLegs = newWidth;
}

float L2Norm(OutlinePair signal, int startOffset,
        OutlinePair model,float topToBottom)
{
 float *top1,*top2,*bottom1,*bottom2;
 int i,overlap;
 float sum;
 float temp;

 if ((startOffset < 0) ||
   (startOffset + model->numberOfLegs > signal->numberOfLegs))
  DoError("L2Norm: the model must overlap the signal.\n",NULL);

 top1 = signal->top+startOffset;
 top2 = model->top;
 bottom1 = signal->bottom+startOffset;
 bottom2 = model->bottom;

 overlap = signal->numberOfLegs - startOffset;
 if (overlap > model->numberOfLegs)
  overlap = model->numberOfLegs;

 for (i=0,sum=0;i<overlap;++i) {
  temp = *top1++ - *top2++;
  sum += temp * temp * topToBottom;
  temp = *bottom1++ - *bottom2++;
  sum += temp * temp;
 }

 return sum;
}

float L2Compare(OutlinePair o1,OutlinePair o2,float topToBottom)
{
 float slope = (float)o1->width/(float)o2->width;
 if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE))
  return BIG_NUM;
 if (o1->numberOfLegs != NORMAL_LENGTH)
  ResampleOutlinePair(o1,NORMAL_LENGTH/o1->numberOfLegs);
 if (o2->numberOfLegs != NORMAL_LENGTH)
  ResampleOutlinePair(o2,NORMAL_LENGTH/o2->numberOfLegs);
 return L2Norm(o1,0,o2,topToBottom);
}
```

Aug 14 20:54 1991 newMatch.c

```
#include <stdio.h>
#include "mylib.h"
#include "misc.h"
#include "types.h"
#include "dict.h"
#include "newMatch.h"

#define MAX_SIGNAL_LENGTH (800)
#define MAX_SLOPE (2.0)
#define BIG_NUM (10e20)

typedef enum {NONE,LEFT,DOWN,DOWNLEFT,D1L1,D2L1,D1L2} Direction;

extern double sqrt(double);
extern double cos(double);
extern double atan(double);
extern int irint(double);

/* Assumes that a represents the model and b represents the unknown.
 * Weights places where the model is lower than the unknown more than
 * cases where the model is higher than the unknown. The idea here is
 * that valleys can be filled in by bleeding together, but that noise
 * can rarely make a contour be too tall for extended periods.
 */
float hillToValley = 1.0;
inline float SquareDifference(float a,float b)
{
 float temp = a-b;
 if (temp<0)
  return temp*temp;
 else
  return temp*temp*hillToValley*hillToValley;
/* return (a-b)*(a-b); */
}

inline float FMax(float a,float b)
{
 if (a>b)
  return a;
 else
  return b;
}

inline float FMin(float a,float b)
{
 if (a<b)
  return a;
 else
  return b;
}

inline int IMax(int a,int b)
```

```
{
 if (a>b)
  return a;
 else
  return b;
}

inline int IMin(int a,int b)
{
 if (a<b)
  return a;
 else
  return b;
}

float NewMatch(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
          float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth,
          float topToBottom)
{
 float costs0[MAX_SIGNAL_LENGTH+1];
 float costs1[MAX_SIGNAL_LENGTH+1];
 int i,j,start,end,bandWidth,shift;
 int realStart,realEnd,center,oldEnd;
 float slope,angle;
 float *a1c,*a2c,*cd,*cl,*cdl,*dc;
 float oldCost,b1v,b2v,returnVal;

 if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
  DoError("NewMatch: maximum signal length exceeded.\n",NULL);

 slope = (float)aLength/(float)bLength;

 if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
  return BIG_NUM;
 }
 angle = atan(slope);
 bandWidth = irint(normalBandWidth/cos(angle));
 center = 0;
 realStart = center-bandWidth/2;
 realEnd = realStart+bandWidth;
 end = FMin(realEnd,aLength);

 a1c = a1; /* a1 cursor */
 a2c = a2; /* a2 cursor */
 b1v = *b1; /* b1 value */
 b2v = *b2; /* b2 value */
 dc = costs0;
 *dc++ = BIG_NUM;
 oldCost = *dc++ =
SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);

 for (j=1;j<end;++j)
  oldCost = *dc++ =
oldCost+SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);
```

```
  for (i=1;i<bLength;++i) {
   /* Compute new center of band */
   center = irint(slope*i);
   realStart = center-bandWidth/2;
   realEnd = realStart+bandWidth;
   start = FMax(realStart,0);
   oldEnd = end;
   end = FMin(realEnd,aLength);
   shift = end-oldEnd;

   /* put large numbers where bands don't overlap */
   for (j=0;j<shift;++j)
    *dc++ = BIG_NUM;

   a1c = a1+start; /* a1 cursor */
   a2c = a2+start; /* a2 cursor */
   b1v = *(b1+i); /* b1 value */
   b2v = *(b2+i); /* b2 value */
   if (i&1) {
    cd = costs1+start-1+1; /* cursor down ??? What about -1??? */
    cdl = costs0+start-1+1; /* cursor down left */
    cl = costs0+start+1; /* cursor left */
    dc = costs1+start+1; /* destination cursor */
   }
   else {
    cd = costs0+start-1+1; /* cursor down */
    cdl = costs1+start-1+1; /* cursor down left */
    cl = costs1+start+1; /* cursor left */
    dc = costs0+start+1; /* destination cursor */
   }
   *cd = BIG_NUM;
   for (j=start;j<end;++j) {
    float down,left,downLeft,rest;
    down = *cd++ + rest;
    left = *cl++ + rest;
    downLeft = *cdl++ + rest*centerWeight;
    rest = SquareDifference(*a1c++,b1v)*topToBottom+
        SquareDifference(*a2c++,b2v);
    *dc++ = FMin(FMin(down,left),downLeft);
   }
  }

  i--;
  if (i&1)
   dc = costs1+aLength-1+1;
  else
   dc = costs0+aLength-1+1;
  returnVal = *dc;

  if (lengthNormalize)
   return returnVal/sqrt(aLength*aLength+bLength*bLength);
  else
   return returnVal;
 }
```

```
float SepMatch(float *a1,int aLength,float *b1,int bLength,
         float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth)
{
 float costs0[MAX_SIGNAL_LENGTH+1];
 float costs1[MAX_SIGNAL_LENGTH+1];
 int i,j,start,end,bandWidth,shift;
 int realStart,realEnd,center,oldEnd;
 float slope,angle;
 float *a1c,*cd,*cl,*cdl,*dc;
 float oldCost,b1v,returnVal;

 if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
  DoError("NewMatch: maximum signal length exceeded.\n",NULL);

 slope = (float)aLength/(float)bLength;

 if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
  return BIG_NUM;
 }
 angle = atan(slope);
 bandWidth = irint(normalBandWidth/cos(angle));
 center = 0;
 realStart = center-bandWidth/2;
 realEnd = realStart+bandWidth;
 end = FMin(realEnd,aLength);

 a1c = a1; /* a1 cursor */
 b1v = *b1; /* b1 value */
 dc = costs0;
 *dc++ = BIG_NUM;
 oldCost = *dc++ = SquareDifference(*a1c++,b1v);

 for (j=1;j<end;++j)
  oldCost = *dc++ = oldCost+SquareDifference(*a1c++,b1v);

 for (i=1;i<bLength;++i) {
  /* Compute new center of band */
  center = irint(slope*i);
  realStart = center-bandWidth/2;
  realEnd = realStart+bandWidth;
  start = FMax(realStart,0);
  oldEnd = end;
  end = FMin(realEnd,aLength);
  shift = end-oldEnd;

  /* put large numbers where bands don't overlap */
  for (j=0;j<shift;++j)
   *dc++ = BIG_NUM;

  a1c = a1+start; /* a1 cursor */
  b1v = *(b1+i); /* b1 value */
  if (i&1) {
   cd = costs1+start-1+1; /* cursor down ??? What about -1??? */
   cdl = costs0+start-1+1; /* cursor down left */
```

```
    cl = costs0+start+1; /* cursor left */
    dc = costs1+start+1; /* destination cursor */
   }
   else {
    cd = costs0+start-1+1; /* cursor down */
    cdl = costs1+start-1+1; /* cursor down left */
    cl = costs1+start+1; /* cursor left */
    dc = costs0+start+1; /* destination cursor */
   }
   *cd = BIG_NUM;
   for (j=start;j<end;++j) {
    float down,left,downLeft,rest;
    down = *cd++ + rest;
    left = *cl++ + rest;
    downLeft = *cdl++ + rest*centerWeight;
    rest = SquareDifference(*a1c++,b1v);
    *dc++ = FMin(FMin(down,left),downLeft);
   }
  }

  i--;
  if (i&1)
   dc = costs1+aLength-1+1;
  else
   dc = costs0+aLength-1+1;
  returnVal = *dc;

  if (lengthNormalize)
   return returnVal/sqrt(aLength*aLength+bLength*bLength);
  else
   return returnVal;
 }

#define WIDTH (800)
#define H_MARGIN (20)
#define V_MARGIN (40)
#define H_SPACING (20)
#define V_SPACING (100) /* Must be greater than 2*X_HEIGHT */
#define X_HEIGHT (17)
void DrawVLine(Picture pict,int x,int yt,int yb)
{
 int i;
 for (i=yt;i<yb;++i)
  WritePixel(pict,x,i,1);
}

void DrawOutline(Picture pict,int numberOfLegs,float *tops,float *bottoms,int x,int y)
{
 int i,top,bottom;
 for (i=0;i<numberOfLegs;++i) {
  top = irint(-*(tops+i)*X_HEIGHT);
  bottom = irint(*(bottoms+i)*X_HEIGHT+X_HEIGHT);
  DrawVLine(pict,i+x,top+y,bottom+y);
 }
}
```

```
void PrintPath(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
            Direction path[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH],int i,
            float returnVal,
            FILE *pathFP)
{
 int x,y,j;
 int length = 0;
 int index = 0;
 float newTop1[MAX_SIGNAL_LENGTH],newBottom1[MAX_SIGNAL_LENGTH];
 float newTop2[MAX_SIGNAL_LENGTH],newBottom2[MAX_SIGNAL_LENGTH];

 y = i;
 x = aLength-1;
 while (path[y][x]!=NONE) {
  switch (path[y][x]) {
  case DOWN:
   x--;
   break;
  case LEFT:
   y--;
   break;
  case D1L1:
  case DOWNLEFT:
   x--;
   y--;
   break;
  case D2L1:
   x-=2;
   y--;
   break;
  case D1L2:
   x--;
   y-=2;
   break;
  default:
   DoError("NewMatchAndPath: Internal error - bad case.\n",NULL);
  }
  ++length;
 }

 y = i;
 x = aLength-1;
 while (path[y][x]!=NONE) {
  if (index>=MAX_SIGNAL_LENGTH)
   DoError("NewMatchAddPath: warped signal is too long.\n",NULL);
  newTop1[length-index] = a1[x];
  newBottom1[length-index] = a2[x];
  newTop2[length-index] = b1[y];
  newBottom2[length-index] = b2[y];
  switch (path[y][x]) {
  case DOWN:
   x--;
   break;
  case LEFT:
```

```
      y--;
      break;
     case D1L1:
     case DOWNLEFT:
      x--;
      y--;
      break;
     case D2L1:
      x-=2;
      y--;
      break;
     case D1L2:
      x--;
      y-=2;
      break;
     default:
      DoError("NewMatchAndPath: Internal error - bad case.\n",NULL);
     }
     ++index;
    }
    if (index>=MAX_SIGNAL_LENGTH)
     DoError("NewMatchAddPath: warped signal is too long.\n",NULL);
    newTop1[length-index] = a1[x];
    newBottom1[length-index] = a2[x];
    newTop2[length-index] = b1[y];
    newBottom2[length-index] = b2[y];
    ++index;

    for (j=0;j<index;++j)
     fprintf(pathFP,"%d %f\n",j,newTop1[j]);
    fprintf(pathFP,"\"top1\n\n");

    for (j=0;j<index;++j)
     fprintf(pathFP,"%d %f\n",j,newTop2[j]);
    fprintf(pathFP,"\"top2\n\n");

    for (j=0;j<index;++j)
     fprintf(pathFP,"%d %f\n",j,-newBottom1[j]);
    fprintf(pathFP,"\"bottom1\n\n");

    for (j=0;j<index;++j)
     fprintf(pathFP,"%d %f\n",j,-newBottom2[j]);
    fprintf(pathFP,"\"bottom2\n\n");

    {
     Picture pict;
     pict =
new_pict(IMax(index,IMax(aLength,bLength))*2+H_SPACING+H_MARGIN*2,V_MARGIN*
2+2*V_SPACING,1);
     DrawOutline(pict,aLength,a1,a2,H_MARGIN,V_MARGIN);
     DrawOutline(pict,bLength,b1,b2,H_MARGIN+aLength+H_SPACING,V_MARGIN);

     DrawOutline(pict,index,newTop1,newBottom1,H_MARGIN,V_MARGIN+V_SPACING);
```

```
DrawOutline(pict,index,newTop2,newBottom2,H_MARGIN+index+H_SPACING,V_MARGIN+V_SPACING);
  DrawOutline(pict,index,newTop2,newBottom2,H_MARGIN,V_MARGIN+V_SPACING*2);
  write_pict("out.pict",pict);
 }

 {
  float checksum;
  fprintf(pathFP,"%d %f\n",0,checksum);
  for (j=0,checksum=0;j<index;++j) {
   checksum+=SquareDifference(newTop1[j],newTop2[j])+
       SquareDifference(newBottom1[j],newBottom2[j]);
   fprintf(pathFP,"%d %f\n",j,checksum);
  }
  printf("checksum, score = %6.2f, %6.2f\n",checksum,returnVal);
 }
}



float NewMatchAndPath(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
            float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth,
            float topToBottom,FILE *pathFP)
{
 Direction path[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH],*pc;
 int x,y;
 float costs0[MAX_SIGNAL_LENGTH+1];
 float costs1[MAX_SIGNAL_LENGTH+1];
 int i,j,start,end,bandWidth,shift;
 int realStart,realEnd,center,oldEnd;
 float slope,angle;
 float *a1c,*a2c,*cd,*cl,*cdl,*dc;
 float oldCost,b1v,b2v,returnVal;

 if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
  DoError("NewMatch: maximum signal length exceeded.\n",NULL);

 slope = (float)aLength/(float)bLength;

 if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
  return BIG_NUM;
 }
 angle = atan(slope);
 bandWidth = irint(normalBandWidth/cos(angle));
 center = 0;
 realStart = center-bandWidth/2;
 realEnd = realStart+bandWidth;
 end = FMin(realEnd,aLength);

 a1c = a1;  /* a1 cursor */
 a2c = a2;  /* a2 cursor */
 b1v = *b1; /* b1 value */
 b2v = *b2; /* b2 value */
```

```
  dc = costs0;
  pc = &(path[0][0]);
  *dc++ = BIG_NUM;
  oldCost = *dc++ =
SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);
  *pc++ = NONE;

  for (j=1;j<end;++j) {
    oldCost = *dc++ =
oldCost+SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);
    *pc++ = DOWN;
  }

#ifdef foo
  printf("%6d ",0);
  for (j=0;j<end;++j)
    printf("%6.2f ",costs0[j+1]);
#endif

  for (i=1;i<bLength;++i) {
    /* Compute new center of band */
    center = irint(slope*i);
    realStart = center-bandWidth/2;
    realEnd = realStart+bandWidth;
    start = FMax(realStart,0);
    oldEnd = end;
    end = FMin(realEnd,aLength);
    shift = end-oldEnd;

    /* put large numbers where bands don't overlap */
    for (j=0;j<shift;++j) {
/*    printf("%6.2f ",BIG_NUM); */
      *dc++ = BIG_NUM;
    }
/*  printf("\n%6d ",i); */

    a1c = a1+start;  /* a1 cursor */
    a2c = a2+start;  /* a2 cursor */
    b1v = *(b1+i); /* b1 value */
    b2v = *(b2+i); /* b2 value */
    pc = &(path[i][start]);
    if (i&1) {
      cd = costs1+start-1+1;  /* cursor down ??? What about -1??? */
      cdl = costs0+start-1+1; /* cursor down left */
      cl = costs0+start+1;  /* cursor left */
      dc = costs1+start+1;  /* destination cursor */
    }
    else {
      cd = costs0+start-1+1;  /* cursor down */
      cdl = costs1+start-1+1; /* cursor down left */
      cl = costs1+start+1;  /* cursor left */
      dc = costs0+start+1;  /* destination cursor */
    }
    *cd = BIG_NUM;
    for (j=start;j<end;++j) {
```

```
    float down,left,downLeft,rest;
    rest = SquareDifference(*a1c++,b1v)*topToBottom +
SquareDifference(*a2c++,b2v);
    down = *cd++ + rest;
    left = *cl++ + rest;
    downLeft = *cdl++ + rest*centerWeight;

    if (down < left)
        if (down < downLeft) {
/*          printf("%6.2f ",down); */
            *dc++ = down;
            *pc++ = DOWN;
        }
        else {
/*          printf("%6.2f ",downLeft); */
            *dc++ = downLeft;
            *pc++ = DOWNLEFT;
        }
    else
        if (downLeft < left) {
/*          printf("%6.2f ",downLeft); */
            *dc++ = downLeft;
            *pc++ = DOWNLEFT;
        }
        else {
/*          printf("%6.2f ",left); */
            *dc++ = left;
            *pc++ = LEFT;
        }
   }
  }

  i--;
  if (i&1)
   dc = costs1+aLength-1+1;
  else
   dc = costs0+aLength-1+1;
  returnVal = *dc;

#ifdef foo
  if (!doPath) {
   y = i;
   x = aLength-1;
   while (path[y][x]!=NONE) {
    switch (path[y][x]) {
    case DOWN:
        x--;
        break;
    case LEFT:
        y--;
        break;
    case DOWNLEFT:
        x--;
        y--;
        break;
```

```
      default:
        DoError("NewMatchAndPath: Internal error - bad case.\n",NULL);
      }
      fprintf(pathFP,"%d %d\n",x,y);
    }
    fprintf(pathFP,"%d %d\n",x,y);
  }
  else {}
#endif
  PrintPath(a1,a2,aLength,b1,b2,bLength,path,i,returnVal,pathFP);

  if (lengthNormalize)
    return returnVal/sqrt(aLength*aLength+bLength*bLength);
  else
    return returnVal;
}


float SlopeCMatch(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
              float centerWeight,BOOLEAN lengthNormalize,float topToBottom)
{
  float costs0[MAX_SIGNAL_LENGTH+2];
  float costs1[MAX_SIGNAL_LENGTH+2];
  float costs2[MAX_SIGNAL_LENGTH+2];
  float slope,minVal;
  int i,j;
  int bottom,top;
  float *cd1l1,*cd2l1,*cd1l2;
  float *a1c,*a2c,*cd,*cl,*cdl,*dc;
  float b1v,b2v,returnVal;

/* printf("sc:\n"); */

  if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
    DoError("SlopeCMatch: maximum signal length exceeded.\n",NULL);

  slope = (float)aLength/(float)bLength;
  if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
    return BIG_NUM;
  }

  for (i=0;i<aLength+2;++i) {
    costs2[i] = BIG_NUM;
    costs1[i] = BIG_NUM;
    costs0[i] = BIG_NUM;
  }

  costs0[2] = SquareDifference(*a1,*b1)*topToBottom+SquareDifference(*a2,*b2);

  for (i=1;i<bLength;++i) {
    bottom = IMax(i/2,2*i+aLength-2*bLength);
    top = IMin(2*i,i/2+aLength-bLength/2)+1;

    a1c = a1+bottom; /* a1 cursor */
    a2c = a2+bottom; /* a2 cursor */
```

```
        b1v = *(b1+i); /* b1 value */
        b2v = *(b2+i); /* b2 value */

        switch (i%3) {
        case 0:
         dc = costs0+bottom-2+2;
         cd2l1 = costs2+bottom-2+2;
         cd1l2 = costs1+bottom-1+2;
         cd1l1 = costs2+bottom-1+2;
         break;
        case 1:
         dc = costs1+bottom-2+2;
         cd2l1 = costs0+bottom-2+2;
         cd1l2 = costs2+bottom-1+2;
         cd1l1 = costs0+bottom-1+2;
         break;
        case 2:
         dc = costs2+bottom-2+2;
         cd2l1 = costs1+bottom-2+2;
         cd1l2 = costs0+bottom-1+2;
         cd1l1 = costs1+bottom-1+2;
         break;
        }
        *dc++ = BIG_NUM;
        *dc++ = BIG_NUM;
        for (j=bottom;j<top;++j) {
         float d2l1,d1l2,d1l1,rest;
         rest = SquareDifference(*a1c++,b1v)*topToBottom+
              SquareDifference(*a2c++,b2v);
         d1l1 = *cd1l1++ + rest*centerWeight;
         d1l2 = *cd1l2++ + rest;
         d2l1 = *cd2l1++ + rest;

         *dc++ = FMin(FMin(d1l1,d2l1),d1l2);
        }

        switch (i%3) {
        case 0:
         dc = costs0;
         break;
        case 1:
         dc = costs1;
         break;
        case 2:
         dc = costs2;
         break;
        }

      #ifdef foo
        minVal = BIG_NUM;
        printf("%6d: ",i);
        for (j=0;j<aLength+2;++j) {
         if (*dc <= minVal)
              minVal = *dc;
         if (*dc++ >= BIG_NUM)
```

```
            printf(" ");
        else
            printf("*");
      }
      printf("    %6.2f\n",minVal);
#endif
    }

    --i;
    switch (i%3) {
    case 0:
      dc = costs0;
      break;
    case 1:
      dc = costs1;
      break;
    case 2:
      dc = costs2;
      break;
    }
    returnVal = *(dc+aLength-1+2);

    if (lengthNormalize)
      return returnVal/sqrt(aLength*aLength+bLength*bLength);
    else
      return returnVal;
}


float SepSlopeCMatch(float *a1,int aLength,float *b1,int bLength,
                float centerWeight,BOOLEAN lengthNormalize)
{
  float costs0[MAX_SIGNAL_LENGTH+2];
  float costs1[MAX_SIGNAL_LENGTH+2];
  float costs2[MAX_SIGNAL_LENGTH+2];
  float slope,minVal;
  int i,j;
  int bottom,top;
  float *cd1l1,*cd2l1,*cd1l2;
  float *a1c,*cd,*cl,*cdl,*dc;
  float b1v,returnVal;

  if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
    DoError("SlopeCMatch: maximum signal length exceeded.\n",NULL);

  slope = (float)aLength/(float)bLength;
  if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
    return BIG_NUM;
  }

  for (i=0;i<aLength+2; ++i) {
    costs2[i] = BIG_NUM;
    costs1[i] = BIG_NUM;
    costs0[i] = BIG_NUM;
  }
```

```
 costs0[2] = SquareDifference(*a1,*b1);

 for (i=1;i<bLength;++i) {
  bottom = IMax(i/2,2*i+aLength-2*bLength);
  top = IMin(2*i,i/2+aLength-bLength/2)+1;

  a1c = a1+bottom; /* a1 cursor */
  b1v = *(b1+i); /* b1 value */

  switch (i%3) {
  case 0:
   dc = costs0+bottom-2+2;
   cd2l1 = costs2+bottom-2+2;
   cd1l2 = costs1+bottom-1+2;
   cd1l1 = costs2+bottom-1+2;
   break;
  case 1:
   dc = costs1+bottom-2+2;
   cd2l1 = costs0+bottom-2+2;
   cd1l2 = costs2+bottom-1+2;
   cd1l1 = costs0+bottom-1+2;
   break;
  case 2:
   dc = costs2+bottom-2+2;
   cd2l1 = costs1+bottom-2+2;
   cd1l2 = costs0+bottom-1+2;
   cd1l1 = costs1+bottom-1+2;
   break;
  }
  *dc++ = BIG_NUM;
  *dc++ = BIG_NUM;
  for (j=bottom;j<top;++j) {
   float d2l1,d1l2,d1l1,rest;
   rest = SquareDifference(*a1c++,b1v);
   d1l1 = *cd1l1++ + rest*centerWeight;
   d1l2 = *cd1l2++ + rest;
   d2l1 = *cd2l1++ + rest;

   *dc++ = FMin(FMin(d1l1,d2l1),d1l2);
  }

  switch (i%3) {
  case 0:
   dc = costs0;
   break;
  case 1:
   dc = costs1;
   break;
  case 2:
   dc = costs2;
   break;
  }
 }
```

```
  --i;
  switch (i%3) {
  case 0:
   dc = costs0;
   break;
  case 1:
   dc = costs1;
   break;
  case 2:
   dc = costs2;
   break;
  }
  returnVal = *(dc+aLength-1+2);

  if (lengthNormalize)
   return returnVal/sqrt(aLength*aLength+bLength*bLength);
  else
   return returnVal;
 }


 float SlopeCMatchAndPath(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
                  float centerWeight,BOOLEAN lengthNormalize,float topToBottom,
                  FILE *pathFP)
 {
  Direction path[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH],*pc;
  float costs0[MAX_SIGNAL_LENGTH+2];
  float costs1[MAX_SIGNAL_LENGTH+2];
  float costs2[MAX_SIGNAL_LENGTH+2];
  float slope,minVal;
  int i,j;
  int bottom,top;
  float *cd1l1,*cd2l1,*cd1l2;
  float *a1c,*a2c,*cd,*cl,*cdl,*dc;
  float b1v,b2v,returnVal;

 /* printf("sc:\n"); */

  if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
   DoError("SlopeCMatch: maximum signal length exceeded.\n",NULL);

  slope = (float)aLength/(float)bLength;
  if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
   return BIG_NUM;
  }

  for (i=0;i<aLength+2;++i) {
   costs2[i] = BIG_NUM;
   costs1[i] = BIG_NUM;
   costs0[i] = BIG_NUM;
  }

  pc = &(path[0][0]);
  *pc++ = NONE;
  costs0[2] = SquareDifference(*a1,*b1)*topToBottom+SquareDifference(*a2,*b2);
```

```
    for (i=1;i<bLength;++i) {
     bottom = IMax(i/2,2*i+aLength-2*bLength);
     top = IMin(2*i,i/2+aLength-bLength/2)+1;

     a1c = a1+bottom;  /* a1 cursor */
     a2c = a2+bottom;  /* a2 cursor */
     b1v = *(b1+i); /* b1 value */
     b2v = *(b2+i); /* b2 value */

     switch (i%3) {
     case 0:
      dc = costs0+bottom-2+2;
      cd2l1 = costs2+bottom-2+2;
      cd1l2 = costs1+bottom-1+2;
      cd1l1 = costs2+bottom-1+2;
      break;
     case 1:
      dc = costs1+bottom-2+2;
      cd2l1 = costs0+bottom-2+2;
      cd1l2 = costs2+bottom-1+2;
      cd1l1 = costs0+bottom-1+2;
      break;
     case 2:
      dc = costs2+bottom-2+2;
      cd2l1 = costs1+bottom-2+2;
      cd1l2 = costs0+bottom-1+2;
      cd1l1 = costs1+bottom-1+2;
      break;
     }
     *dc++ = BIG_NUM;
     *dc++ = BIG_NUM;
     pc = &(path[i][bottom]);
     for (j=bottom;j<top;++j) {
      float d2l1,d1l2,d1l1,rest;

      rest = SquareDifference(*a1c++,b1v)*topToBottom+
          SquareDifference(*a2c++,b2v);
      d1l1 = *cd1l1++ + rest*centerWeight;
      d1l2 = *cd1l2++ + rest;
      d2l1 = *cd2l1++ + rest;

      if (d1l1<d1l2)
          if (d1l1<d2l1) {
            *dc++ = d1l1;
            *pc++ = D1L1;
          }
          else {
            *dc++ = d2l1;
            *pc++ = D2L1;
          }
      else
          if (d1l2<d2l1) {
            *dc++ = d1l2;
            *pc++ = D1L2;
```

```
                }
                else {
                 *dc++ = d2l1;
                 *pc++ = D2L1;
                }
            }

            switch (i%3) {
            case 0:
             dc = costs0;
             break;
            case 1:
             dc = costs1;
             break;
            case 2:
             dc = costs2;
             break;
            }
            minVal = BIG_NUM;
            printf("%6d: ",i);
            for (j=0;j<aLength+2;++j) {
             if (*dc <= minVal)
                 minVal = *dc;
             if (*dc++ >= BIG_NUM)
                 printf(" ");
             else
                 printf("*");
            }
            printf("    %6.2f\n",minVal);
           }

           --i;
           switch (i%3) {
           case 0:
            dc = costs0;
            break;
           case 1:
            dc = costs1;
            break;
           case 2:
            dc = costs2;
            break;
           }
           returnVal = *(dc+aLength-1+2);

           PrintPath(a1,a2,aLength,b1,b2,bLength,path,i,returnVal,pathFP);

           if (lengthNormalize)
            return returnVal/sqrt(aLength*aLength+bLength*bLength);
           else
            return returnVal;
          }
```

Aug 2 02:29 1991 recogDesc.c

```
#include <stdio.h>
#include "mylib.h"
#include "types.h"
#include "dict.h"
#include "diff.h"

#define BIG_NUM (10e10)
#define MAX_WORDS (100)
#define MAX_FONTS (10)

extern double sqrt(double);

float CompareNumericDescriptors(float *a,float *b,int length)
{
 int i;
 float sum;
 for (i=0,sum=0;i<length;++i) {
  sum += (*a-*b)*(*a-*b);
  ++a;
  ++b;
 }
 return sqrt(sum);
}

float *ComputeNumericDescriptor(int modelIndex,Dictionary models,
                              Dictionary *fonts,int numberOfFonts,int numberOfWords,
                              DiffDescriptor dd,
                              float *sd,float *avg)
{
 float *d;
 int i,j;
 float temp;
 float sumxx[MAX_WORDS];
 float sdev[MAX_WORDS];
 float sumsdev,sumscore;

 if ((d = (float *)calloc(numberOfWords,sizeof(float)))==NULL)
  DoError("ComputeNumericDescriptor: cannot allocate space.\n",NULL);
 for (j=0;j<numberOfWords;++j)
  sumxx[j]=0;
 for (i=0;i<numberOfFonts;++i)
  for (j=0;j<numberOfWords;++j) {
   temp=DiffPair(*(models->outlines+modelIndex),*(fonts[i]->outlines+j),dd);
   if (temp < BIG_NUM) {
       d[j]+=temp;
       sumxx[j]+=temp*temp;
   }
  }

 if (numberOfFonts > 1) {
  float sum,minsdev,maxsdev;
  for (j=0;j<numberOfWords;++j)
```

```
    sdev[j] = sqrt((numberOfFonts*sumxx[j]-d[j]*d[j])/numberOfFonts/(numberOfFonts-1));
   for (j=0,sumsdev=0,sumscore=0;j<numberOfWords;++j) {
    sumsdev += sdev[j];
    sumscore += d[j];
   }
   *sd = sumsdev/numberOfWords;
   *avg = sumscore/numberOfWords;
  }

  for (j=0;j<numberOfWords;++j)
   d[j]/=numberOfFonts;

  return d;
}

typedef struct {
 float score;
 int x;
 int y;
} *CompareTuple,CompareTupleBody;

int TupleLessThan(CompareTuple *x,CompareTuple *y)
{
 if ((*x)->score == (*y)->score)
  return 0;
 else if ((*x)->score < (*y)->score)
  return -1;
 else
  return 1;
}


void DoDescriptors(Dictionary models,char *modelName,char **wordNames,
                int numberOfFonts,Dictionary *fonts,char **fontNames,
                int numberOfWords,DiffDescriptor dd)
{
 float *descriptors[MAX_WORDS];
 int classes[MAX_WORDS][MAX_WORDS];
 float sdev[MAX_WORDS],avg[MAX_WORDS];
 CompareTupleBody tuples[MAX_WORDS*MAX_WORDS];
 CompareTuple scores[MAX_WORDS*MAX_WORDS];
 int i,x,y,j;
 int count;
/* float threshold = 0.22; */
 float threshold = 0.42;

 for (i=0;i<numberOfWords;++i) {
  descriptors[i] =
ComputeNumericDescriptor(i,models,fonts,numberOfFonts,numberOfWords,dd,
                               sdev+i,avg+i);
  fprintf(stdout,"%s: %6.4f %6.4f\n",wordNames[i],avg[i],sdev[i]);
 }
 fprintf(stdout,"\n\n");

 for (y=0;y<numberOfWords;++y)
```

```
  for (x=0;x<numberOfWords;++x)
   classes[y][x] =
(CompareNumericDescriptors(descriptors[y],descriptors[x],numberOfWords)
                 < threshold);


#ifdef foo
 for (y=0,i=0;y<numberOfWords;++y)
  for (x=0;x<y;++x) {
   CompareTuple temp;
/*
   temp = (CompareTuple)calloc(1,sizeof(CompareTupleBody));
   if (temp==NULL)
       DoError(": cannot allocate space.\n",NULL);
*/
   temp = tuples+i;
   temp->score =
CompareNumericDescriptors(descriptors[y],descriptors[x],numberOfWords);
   temp->x = x;
   temp->y = y;
   scores[i] = temp;
   ++i;
  }
 qsort(scores,i,sizeof(CompareTuple),TupleLessThan);

 for (j=0;j<i;++j)
  fprintf(stdout,"(%s,%s):
%f\n",wordNames[scores[j]->y],wordNames[scores[j]->x],scores[j]->score);
#endif

 fprintf(stdout,"\n\n");
 for (i=0;i<numberOfWords;++i) {
  CompareTuple temp;
  float *thisDesc;
  float junk;
  thisDesc =
ComputeNumericDescriptor(i,models,&models,1,numberOfWords,dd,&junk,&junk);
  for (j=0;j<numberOfWords;++j) {
   temp = tuples+j;
   temp->score = CompareNumericDescriptors(thisDesc,descriptors[j],numberOfWords);
   temp->y = i;
   temp->x = j;
   scores[j] = temp;
  }
  qsort(scores,numberOfWords,sizeof(CompareTuple),TupleLessThan);

  fprintf(stdout,"%s: ",wordNames[i]);
  for (j=0;j<5&&j<numberOfWords;++j) {
   fprintf(stdout,"%s ",wordNames[scores[j]->x]);
   if (scores[j]->x == i)
       break;
  }
  if (scores[j]->x==i)
   fprintf(stdout,"\n");
  else {
```

```
   for (;j<numberOfWords;++j)
       if (scores[j]->x==i)
        break;
   fprintf(stdout," (%d more)\n",j-5);
  }

  fprintf(stdout," ");
  count = 0;
  for (j=0;j<numberOfWords;++j)
   if (classes[scores[0]->x][j]) {
       fprintf(stdout,"%s ",wordNames[j]);
       ++count;
       if (count > 5)
        break;
   }
  if (j<numberOfWords) {
   for (count=0;j<numberOfWords;++j)
       if (classes[scores[0]->x][j])
        ++count;
   fprintf(stdout," (% more)\n",count);
  }
  else
   fprintf(stdout,"\n");

  free(thisDesc);
 }
}

void main(int argc,char **argv)
{
 char *listFile;
 Dictionary models;
 char *modelName;
 int numberOfFonts;
 Dictionary fonts[MAX_FONTS];
 char *fontNames[MAX_FONTS];
 char *wordNames[MAX_WORDS];
 int numberOfWords;
 float centerWeight;
 int normalBandWidth;
 BOOLEAN
lengthNormalize,useL2,slopeConstrain,warp,topToBottomOption,hillToValleyOption;
 BOOLEAN separate;
 float topToBottom,hillToValleyLocal;
 FILE *listfp;
 int i,x,y;
 DiffDescriptorBody dd;

 centerWeight = 1.0;
 normalBandWidth = 20;
 topToBottom = 1.0;
 hillToValleyLocal = 1.0;
 DefArg("%s","listFile",&listFile);
 DefOption("-L2","-L2",&useL2);
 DefOption("-slopeConstrain %f","-slopeConstrain <center weight>",
```

```
        &slopeConstrain,&centerWeight);
 DefOption("-warp %f %d","-warp <center weight> <band width>",
        &warp,&centerWeight,&normalBandWidth);
 DefOption("-separate","-separate",&separate);
 DefOption("-normalize","-normalize",&lengthNormalize);
 DefOption("-topToBottom %f","-topToBottom <ratio>",&topToBottomOption,&topToBottom);
 DefOption("-hillToValley %f","-hillToValley <ratio>",&hillToValleyOption,&hillToValleyLocal);
 ScanArgs(argc,argv);

 if ((listfp = fopen(listFile,"r"))==NULL)
  DoError("Error opening file %s.\n",listFile);

/* Read in the number of words in each dictionary */
 numberOfWords = ReadInt(listfp);
 if (numberOfWords > MAX_WORDS)
  DoError("%s: too many words.\n",argv[0]);

/* Read in the words */
 for (i=0;i<numberOfWords;++i) {
  wordNames[i] = ReadString(listfp);
 }

/* Read in the model dictionary */
 modelName = ReadString(listfp);
 models = ReadDictionary(modelName);

/* Read in the number of dictionaries */
 numberOfFonts = ReadInt(listfp);
 if (numberOfFonts > MAX_FONTS)
  DoError("%s: too many dictionaries.\n",argv[0]);

/* Read in the dictionaries and their names */
 for (i=0;i<numberOfFonts;++i) {
  fontNames[i] = ReadString(listfp);
  fonts[i] = ReadDictionary(fontNames[i]);
 }

/* Check to see that all dictionaries have the same number of shapes as the specified number of words. */
 for (i=1;i<numberOfFonts;++i)
  if (fonts[i]->numberOfEntries < numberOfWords)
   DoError("Dictionary %s has too few entries.\n",fontNames[i]);
 if (models->numberOfEntries < numberOfWords)
  DoError("Model dictionary has too few of entries.\n",NULL);


 if (useL2) {
  fprintf(stdout,"Using L2 on length normalized shapes.\n");
  dd.diffType = L2;
 }
 else if (slopeConstrain) {
  fprintf(stdout,"Using dynamic time warping with slope contrained to [0.5,2].\n");
  dd.diffType = CONSTRAINED;
```

```
    dd.separate = separate;
    if (separate)
      fprintf(stdout,"Top and bottom warped separately.\n");
    else
      fprintf(stdout,"Top and bottom warped together.\n");
  }
  else {
    fprintf(stdout,"Using dynamic time warping with bandwidth %d.\n",normalBandWidth);
    dd.diffType = WARP;
    dd.bandWidth = normalBandWidth;
    dd.separate = separate;
    if (separate)
      fprintf(stdout,"Top and bottom warped separately.\n");
    else
      fprintf(stdout,"Top and bottom warped together.\n");
  }
  if (!useL2) {
    fprintf(stdout,"Center weight = %f.\n",centerWeight);
    dd.centerWeight = centerWeight;
    if (lengthNormalize) {
      dd.lengthNormalize = TRUE;
      fprintf(stdout,"Scores normalized by signal length.\n");
    }
    else
      dd.lengthNormalize = FALSE;
  }
  dd.hillToValley = hillToValleyLocal;
  dd.topToBottom = topToBottom;
  dd.pathFP = NULL;

  fprintf(stdout,"Words:\n");
  for (i=0;i<numberOfWords;++i)
    fprintf(stdout,"%d: %s\n",i,wordNames[i]);
  fprintf(stdout,"\n");
  fprintf(stdout,"Model font is %s.\n",modelName);
  fprintf(stdout,"Fonts:\n");
  for (i=0;i<numberOfFonts;++i)
    fprintf(stdout,"%d: %s\n",i,fontNames[i]);
  fprintf(stdout,"\n");


DoDescriptors(models,modelName,wordNames,numberOfFonts,fonts,fontNames,numberO
fWords,&dd);
}
```

Jun 18 16:20 1991 resample.c

```
#include <stdio.h>
#include <values.h>
#include <string.h>
#include <floatingpoint.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "dict.h"

void Resample(OutlinePair signal,float factor)
{
 int i,count;
 float pivot;
 float delFactor;
 float *oldTop,*newTop;
 float *oldBottom,*newBottom;
 float *topSPtr,*topDPtr;
 float *bottomSPtr,*bottomDPtr;

 delFactor = 1.0 - factor;
 for (i=0,count=0,pivot=0.0;i<signal->numberOfLegs;++i) {
  if (pivot>=1.0) {
   pivot -= 1.0;
   pivot += delFactor;
  }
  else {
   pivot += delFactor;
   ++count;
  }
 }

 newTop = (float *)calloc(count,sizeof(float));
 newBottom = (float *)calloc(count,sizeof(float));
 if ((newTop==NULL)||(newBottom==NULL))
  DoError("Resample: cannot allocate space.\n",NULL);

 oldTop = signal->top;
 oldBottom = signal->bottom;

 topSPtr = signal->top;
 bottomSPtr = signal->bottom;
 topDPtr = newTop;
 bottomDPtr = newBottom;
 for (i=0,pivot=0.0;i<signal->numberOfLegs;++i) {
  if (pivot>=1.0) {
   pivot -= 1.0;
   pivot += delFactor;
   ++topSPtr;
   ++bottomSPtr;
  }
  else {
   pivot += delFactor;
```

```
      *topDPtr++ = *topSPtr++;
      *bottomDPtr++ = *bottomSPtr++;
     }
    }

   signal->top = newTop;
   signal->bottom = newBottom;
   signal->numberOfLegs = count;

   free(oldTop);
   free(oldBottom);
  }

  void main(int argc,char **argv)
  {
   char *inFile,*outFile;
   float factor;
   int i;
   Dictionary dict;

   if (argc != 4) {
    fprintf(stderr,"Usage:\n");
    fprintf(stderr," %s <input file> <output file> <compression factor>\n",argv[0]);
    fprintf(stderr,"Compresses shapes horizontally.\n");
    exit(-1);
   }

   if ((factor>=1.0)||(factor<0.0))
    DoError("%s: factor must be between 0 and 1.\n",argv[0]);

   inFile = argv[1];
   outFile = argv[2];
   factor = atof(argv[3]);

   dict = ReadDictionary(inFile);

   for (i=0;i<dict->numberOfEntries;++i)
    Resample(*(dict->outlines+i),factor);

   WriteDictionary(dict,outFile);
  }
```

Jul 31 16:48 1991 sepMatch.c

```
#include <stdio.h>
#include "mylib.h"
#include "misc.h"

#define MAX_SIGNAL_LENGTH (800)
#define MAX_SLOPE (2.0)
#define BIG_NUM (10e20)

typedef enum {NONE,LEFT,DOWN,DOWNLEFT,D1L1,D2L1,D1L2} Direction;

extern double sqrt(double);
extern double cos(double);
extern double atan(double);
extern int irint(double);

/* Assumes that a represents the model and b represents the unknown.
 * Weights places where the model is lower than the unknown more than
 * cases where the model is higher than the unknown. The idea here is
 * that valleys can be filled in by bleeding together, but that noise
 * can rarely make a contour be too tall for extended periods.
 */
float hillToValley = 1.0;
inline float SquareDifference(float a,float b)
{
 float temp = a-b;
 if (temp<0)
  return temp*temp;
 else
  return temp*temp*hillToValley*hillToValley;
/* return (a-b)*(a-b); */
}

inline float FMax(float a,float b)
{
 if (a>b)
  return a;
 else
  return b;
}

inline float FMin(float a,float b)
{
 if (a<b)
  return a;
 else
  return b;
}

inline int IMax(int a,int b)
{
 if (a>b)
  return a;
```

```
  else
   return b;
}

inline int IMin(int a,int b)
{
  if (a<b)
   return a;
  else
   return b;
}

float NewMatch(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
          float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth,
          float topToBottom)
{
 float costs0[MAX_SIGNAL_LENGTH+1];
 float costs1[MAX_SIGNAL_LENGTH+1];
 int i,j,start,end,bandWidth,shift;
 int realStart,realEnd,center,oldEnd;
 float slope,angle;
 float *a1c,*a2c,*cd,*cl,*cdl,*dc;
 float oldCost,b1v,b2v,returnVal;

 if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
  DoError("NewMatch: maximum signal length exceeded.\n",NULL);

 slope = (float)aLength/(float)bLength;

 if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
  return BIG_NUM;
 }
 angle = atan(slope);
 bandWidth = irint(normalBandWidth/cos(angle));
 center = 0;
 realStart = center-bandWidth/2;
 realEnd = realStart+bandWidth;
 end = FMin(realEnd,aLength);

 a1c = a1;  /* a1 cursor */
 a2c = a2;  /* a2 cursor */
 b1v = *b1; /* b1 value */
 b2v = *b2; /* b2 value */
 dc = costs0;
 *dc++ = BIG_NUM;
 oldCost = *dc++ =
SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);

 for (j=1;j<end;++j)
  oldCost = *dc++ =
oldCost+SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);

 for (i=1;i<bLength;++i) {
  /* Compute new center of band */
  center = irint(slope*i);
```

```
    realStart = center-bandWidth/2;
    realEnd = realStart+bandWidth;
    start = FMax(realStart,0);
    oldEnd = end;
    end = FMin(realEnd,aLength);
    shift = end-oldEnd;

    /* put large numbers where bands don't overlap */
    for (j=0;j<shift;++j)
     *dc++ = BIG_NUM;

    a1c = a1+start; /* a1 cursor */
    a2c = a2+start; /* a2 cursor */
    b1v = *(b1+i); /* b1 value */
    b2v = *(b2+i); /* b2 value */
    if (i&1) {
     cd = costs1+start-1+1; /* cursor down ??? What about -1??? */
     cdl = costs0+start-1+1; /* cursor down left */
     cl = costs0+start+1; /* cursor left */
     dc = costs1+start+1; /* destination cursor */
    }
    else {
     cd = costs0+start-1+1; /* cursor down */
     cdl = costs1+start-1+1; /* cursor down left */
     cl = costs1+start+1; /* cursor left */
     dc = costs0+start+1; /* destination cursor */
    }
    *cd = BIG_NUM;
    for (j=start;j<end;++j) {
     float down,left,downLeft,rest;
     down = *cd++ + rest;
     left = *cl++ + rest;
     downLeft = *cdl++ + rest*centerWeight;
     rest = SquareDifference(*a1c++,b1v)*topToBottom+
          SquareDifference(*a2c++,b2v);
     *dc++ = FMin(FMin(down,left),downLeft);
    }
   }

   i--;
   if (i&1)
    dc = costs1+aLength-1+1;
   else
    dc = costs0+aLength-1+1;
   returnVal = *dc;

   if (lengthNormalize)
    return returnVal/sqrt(aLength*aLength+bLength*bLength);
   else
    return returnVal;
  }


  void PrintPath(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
           Direction path[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH],int i,
```

```
            float returnVal,
            FILE *pathFP)
{
 int x,y,j;
 int length = 0;
 int index = 0;
 float newTop1[MAX_SIGNAL_LENGTH],newBottom1[MAX_SIGNAL_LENGTH];
 float newTop2[MAX_SIGNAL_LENGTH],newBottom2[MAX_SIGNAL_LENGTH];

 y = i;
 x = aLength-1;
 while (path[y][x]!=NONE) {
  switch (path[y][x]) {
  case DOWN:
   x--;
   break;
  case LEFT:
   y--;
   break;
  case D1L1:
  case DOWNLEFT:
   x--;
   y--;
   break;
  case D2L1:
   x-=2;
   y--;
   break;
  case D1L2:
   x--;
   y-=2;
   break;
  default:
   DoError("NewMatchAndPath: Internal error - bad case.\n",NULL);
  }
  ++length;
 }

 y = i;
 x = aLength-1;
 while (path[y][x]!=NONE) {
  if (index>=MAX_SIGNAL_LENGTH)
   DoError("NewMatchAddPath: warped signal is too long.\n",NULL);
  newTop1[length-index] = a1[x];
  newBottom1[length-index] = a2[x];
  newTop2[length-index] = b1[y];
  newBottom2[length-index] = b2[y];
  switch (path[y][x]) {
  case DOWN:
   x--;
   break;
  case LEFT:
   y--;
   break;
  case D1L1:
```

```
case DOWNLEFT:
 x--;
 y--;
 break;
case D2L1:
 x-=2;
 y--;
 break;
case D1L2:
 x--;
 y-=2;
 break;
default:
 DoError("NewMatchAndPath: Internal error - bad case.\n",NULL);
}
 ++index;
}
if (index>=MAX_SIGNAL_LENGTH)
 DoError("NewMatchAddPath: warped signal is too long.\n",NULL);
newTop1[length-index] = a1[x];
newBottom1[length-index] = a2[x];
newTop2[length-index] = b1[y];
newBottom2[length-index] = b2[y];
++index;

for (j=0;j<index;++j)
 fprintf(pathFP,"%d %f\n",j,newTop1[j]);
fprintf(pathFP,"\"top1\n\n");

for (j=0;j<index;++j)
 fprintf(pathFP,"%d %f\n",j,newTop2[j]);
fprintf(pathFP,"\"top2\n\n");

for (j=0;j<index;++j)
 fprintf(pathFP,"%d %f\n",j,-newBottom1[j]);
fprintf(pathFP,"\"bottom1\n\n");

for (j=0;j<index;++j)
 fprintf(pathFP,"%d %f\n",j,-newBottom2[j]);
fprintf(pathFP,"\"bottom2\n\n");

{
 float checksum;
 fprintf(pathFP,"%d %f\n",0,checksum);
 for (j=0,checksum=0;j<index;++j){
  checksum+=SquareDifference(newTop1[j],newTop2[j])+
      SquareDifference(newBottom1[j],newBottom2[j]);
  fprintf(pathFP,"%d %f\n",j,checksum);
 }
 printf("checksum, score = %6.2f, %6.2f\n",checksum,returnVal);
}
}
```

```
float NewMatchAndPath(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
                float centerWeight,BOOLEAN lengthNormalize,int normalBandWidth,
                float topToBottom,FILE *pathFP)
{
 Direction path[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH],*pc;
 int x,y;
 float costs0[MAX_SIGNAL_LENGTH+1];
 float costs1[MAX_SIGNAL_LENGTH+1];
 int i,j,start,end,bandWidth,shift;
 int realStart,realEnd,center,oldEnd;
 float slope,angle;
 float *a1c,*a2c,*cd,*cl,*cdl,*dc;
 float oldCost,b1v,b2v,returnVal;

 if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
  DoError("NewMatch: maximum signal length exceeded.\n",NULL);

 slope = (float)aLength/(float)bLength;

 if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
  return BIG_NUM;
 }
 angle = atan(slope);
 bandWidth = irint(normalBandWidth/cos(angle));
 center = 0;
 realStart = center-bandWidth/2;
 realEnd = realStart+bandWidth;
 end = FMin(realEnd,aLength);

 a1c = a1;  /* a1 cursor */
 a2c = a2;  /* a2 cursor */
 b1v = *b1; /* b1 value */
 b2v = *b2; /* b2 value */
 dc = costs0;
 pc = &(path[0][0]);
 *dc++ = BIG_NUM;
 oldCost = *dc++ =
SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);
 *pc++ = NONE;

 for (j=1;j<end;++j) {
  oldCost = *dc++ =
oldCost+SquareDifference(*a1c++,b1v)*topToBottom+SquareDifference(*a2c++,b2v);
  *pc++ = DOWN;
 }

#ifdef foo
 printf("%6d ",0);
 for (j=0;j<end;++j)
  printf("%6.2f ",costs0[j+1]);
#endif

 for (i=1;i<bLength;++i) {
  /* Compute new center of band */
  center = irint(slope*i);
```

```
    realStart = center-bandWidth/2;
    realEnd = realStart+bandWidth;
    start = FMax(realStart,0);
    oldEnd = end;
    end = FMin(realEnd,aLength);
    shift = end-oldEnd;

    /* put large numbers where bands don't overlap */
    for (j=0;j<shift;++j) {
/*    printf("%6.2f ",BIG_NUM); */
     *dc++ = BIG_NUM;
    }
/*  printf("\n%6d ",i); */

    a1c = a1+start; /* a1 cursor */
    a2c = a2+start; /* a2 cursor */
    b1v = *(b1+i); /* b1 value */
    b2v = *(b2+i); /* b2 value */
    pc = &(path[i][start]);
    if (i&1) {
     cd = costs1+start-1+1; /* cursor down ??? What about -1??? */
     cdl = costs0+start-1+1; /* cursor down left */
     cl = costs0+start+1; /* cursor left */
     dc = costs1+start+1; /* destination cursor */
    }
    else {
     cd = costs0+start-1+1; /* cursor down */
     cdl = costs1+start-1+1; /* cursor down left */
     cl = costs1+start+1; /* cursor left */
     dc = costs0+start+1; /* destination cursor */
    }
    *cd = BIG_NUM;
    for (j=start;j<end;++j) {
     float down,left,downLeft,rest;
     rest = SquareDifference(*a1c++,b1v)*topToBottom +
SquareDifference(*a2c++,b2v);
     down = *cd++ + rest;
     left = *cl++ + rest;
     downLeft = *cdl++ + rest*centerWeight;

     if (down < left)
         if (down < downLeft) {
/*         printf("%6.2f ",down); */
          *dc++ = down;
          *pc++ = DOWN;
         }
         else {
/*         printf("%6.2f ",downLeft); */
          *dc++ = downLeft;
          *pc++ = DOWNLEFT;
         }
     else
         if (downLeft < left) {
/*         printf("%6.2f ",downLeft); */
          *dc++ = downLeft;
```

```
            *pc++ = DOWNLEFT;
          }
          else {
/*          printf("%6.2f ",left); */
            *dc++ = left;
            *pc++ = LEFT;
          }
     }
   }

   i--;
   if (i&1)
     dc = costs1+aLength-1+1;
   else
     dc = costs0+aLength-1+1;
   returnVal = *dc;

#ifdef foo
   if (!doPath) {
     y = i;
     x = aLength-1;
     while (path[y][x]!=NONE) {
       switch (path[y][x]) {
       case DOWN:
           x--;
           break;
       case LEFT:
           y--;
           break;
       case DOWNLEFT:
           x--;
           y--;
           break;
       default:
           DoError("NewMatchAndPath: Internal error - bad case.\n",NULL);
       }
       fprintf(pathFP,"%d %d\n",x,y);
     }
     fprintf(pathFP,"%d %d\n",x,y);
   }
   else {}
#endif
   PrintPath(a1,a2,aLength,b1,b2,bLength,path,i,returnVal,pathFP);

   if (lengthNormalize)
     return returnVal/sqrt(aLength*aLength+bLength*bLength);
   else
     return returnVal;
}


float SlopeCMatch(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
              float centerWeight,BOOLEAN lengthNormalize,float topToBottom)
{
   float costs0[MAX_SIGNAL_LENGTH+2];
```

```
    float costs1[MAX_SIGNAL_LENGTH+2];
    float costs2[MAX_SIGNAL_LENGTH+2];
    float slope,minVal;
    int i,j;
    int bottom,top;
    float *cd1l1,*cd2l1,*cd1l2;
    float *a1c,*a2c,*cd,*cl,*cdl,*dc;
    float b1v,b2v,returnVal;

   /* printf("sc:\n"); */

    if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
     DoError("SlopeCMatch: maximum signal length exceeded.\n",NULL);

    slope = (float)aLength/(float)bLength;
    if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
     return BIG_NUM;
    }

    for (i=0;i<aLength+2; ++i) {
     costs2[i] = BIG_NUM;
     costs1[i] = BIG_NUM;
     costs0[i] = BIG_NUM;
    }

    costs0[2] = SquareDifference(*a1,*b1)*topToBottom+SquareDifference(*a2,*b2);

    for (i=1;i<bLength; ++i) {
     bottom = IMax(i/2,2*i+aLength-2*bLength);
     top = IMin(2*i,i/2+aLength-bLength/2)+1;

     a1c = a1+bottom; /* a1 cursor */
     a2c = a2+bottom; /* a2 cursor */
     b1v = *(b1+i); /* b1 value */
     b2v = *(b2+i); /* b2 value */

     switch (i%3) {
     case 0:
      dc = costs0+bottom-2+2;
      cd2l1 = costs2+bottom-2+2;
      cd1l2 = costs1+bottom-1+2;
      cd1l1 = costs2+bottom-1+2;
      break;
     case 1:
      dc = costs1+bottom-2+2;
      cd2l1 = costs0+bottom-2+2;
      cd1l2 = costs2+bottom-1+2;
      cd1l1 = costs0+bottom-1+2;
      break;
     case 2:
      dc = costs2+bottom-2+2;
      cd2l1 = costs1+bottom-2+2;
      cd1l2 = costs0+bottom-1+2;
      cd1l1 = costs1+bottom-1+2;
      break;
```

```
  }
  *dc++ = BIG_NUM;
  *dc++ = BIG_NUM;
  for (j=bottom;j<top;++j) {
   float d2l1,d1l2,d1l1,rest;
   rest = SquareDifference(*a1c++,b1v)*topToBottom+
       SquareDifference(*a2c++,b2v);
   d1l1 = *cd1l1++ + rest*centerWeight;
   d1l2 = *cd1l2++ + rest;
   d2l1 = *cd2l1++ + rest;

   *dc++ = FMin(FMin(d1l1,d2l1),d1l2);
  }

  switch (i%3) {
  case 0:
   dc = costs0;
   break;
  case 1:
   dc = costs1;
   break;
  case 2:
   dc = costs2;
   break;
  }

#ifdef foo
  minVal = BIG_NUM;
  printf("%6d: ",i);
  for (j=0;j<aLength+2;++j) {
   if (*dc <= minVal)
       minVal = *dc;
   if (*dc++ >= BIG_NUM)
       printf(" ");
   else
       printf("*");
  }
  printf("   %6.2f\n",minVal);
#endif
 }

 --i;
 switch (i%3) {
 case 0:
  dc = costs0;
  break;
 case 1:
  dc = costs1;
  break;
 case 2:
  dc = costs2;
  break;
 }
 returnVal = *(dc+aLength-1+2);
```

```
  if (lengthNormalize)
   return returnVal/sqrt(aLength*aLength+bLength*bLength);
  else
   return returnVal;
}


float SlopeCMatchAndPath(float *a1,float *a2,int aLength,float *b1,float *b2,int bLength,
                       float centerWeight,BOOLEAN lengthNormalize,float topToBottom,
                       FILE *pathFP)
{
  Direction path[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH],*pc;
  float costs0[MAX_SIGNAL_LENGTH+2];
  float costs1[MAX_SIGNAL_LENGTH+2];
  float costs2[MAX_SIGNAL_LENGTH+2];
  float slope,minVal;
  int i,j;
  int bottom,top;
  float *cd1l1,*cd2l1,*cd1l2;
  float *a1c,*a2c,*cd,*cl,*cdl,*dc;
  float b1v,b2v,returnVal;

/* printf("sc:\n"); */

  if (aLength>MAX_SIGNAL_LENGTH||bLength>MAX_SIGNAL_LENGTH)
   DoError("SlopeCMatch: maximum signal length exceeded.\n",NULL);

  slope = (float)aLength/(float)bLength;
  if ((slope>MAX_SLOPE)||(1/slope>MAX_SLOPE)) {
   return BIG_NUM;
  }

  for (i=0;i<aLength+2; ++i) {
   costs2[i] = BIG_NUM;
   costs1[i] = BIG_NUM;
   costs0[i] = BIG_NUM;
  }

  pc = &(path[0][0]);
  *pc++ = NONE;
  costs0[2] = SquareDifference(*a1,*b1)*topToBottom+SquareDifference(*a2,*b2);

  for (i=1;i<bLength;++i) {
   bottom = IMax(i/2,2*i+aLength-2*bLength);
   top = IMin(2*i,i/2+aLength-bLength/2)+1;

   a1c = a1+bottom; /* a1 cursor */
   a2c = a2+bottom; /* a2 cursor */
   b1v = *(b1+i); /* b1 value */
   b2v = *(b2+i); /* b2 value */

   switch (i%3) {
   case 0:
    dc = costs0+bottom-2+2;
    cd2l1 = costs2+bottom-2+2;
```

```
   cd1l2 = costs1+bottom-1+2;
   cd1l1 = costs2+bottom-1+2;
   break;
  case 1:
   dc = costs1+bottom-2+2;
   cd2l1 = costs0+bottom-2+2;
   cd1l2 = costs2+bottom-1+2;
   cd1l1 = costs0+bottom-1+2;
   break;
  case 2:
   dc = costs2+bottom-2+2;
   cd2l1 = costs1+bottom-2+2;
   cd1l2 = costs0+bottom-1+2;
   cd1l1 = costs1+bottom-1+2;
   break;
  }
  *dc++ = BIG_NUM;
  *dc++ = BIG_NUM;
  pc = &(path[i][bottom]);
  for (j=bottom;j<top;++j) {
   float d2l1,d1l2,d1l1,rest;

   rest = SquareDifference(*a1c++,b1v)*topToBottom+
       SquareDifference(*a2c++,b2v);
   d1l1 = *cd1l1++ + rest*centerWeight;
   d1l2 = *cd1l2++ + rest;
   d2l1 = *cd2l1++ + rest;

   if (d1l1<d1l2)
       if (d1l1<d2l1) {
        *dc++ = d1l1;
        *pc++ = D1L1;
       }
       else {
        *dc++ = d2l1;
        *pc++ = D2L1;
       }
   else
       if (d1l2<d2l1) {
        *dc++ = d1l2;
        *pc++ = D1L2;
       }
       else {
        *dc++ = d2l1;
        *pc++ = D2L1;
       }
  }

  switch (i%3) {
  case 0:
   dc = costs0;
   break;
  case 1:
   dc = costs1;
   break;
```

```
   case 2:
    dc = costs2;
    break;
   }
   minVal = BIG_NUM;
   printf("%6d: ",i);
   for (j=0;j<aLength+2;++j) {
    if (*dc <= minVal)
        minVal = *dc;
    if (*dc++ >= BIG_NUM)
        printf(" ");
    else
        printf("*");
   }
   printf("    %6.2f\n",minVal);
  }

  --i;
  switch (i%3) {
  case 0:
   dc = costs0;
   break;
  case 1:
   dc = costs1;
   break;
  case 2:
   dc = costs2;
   break;
  }
  returnVal = *(dc+aLength-1+2);

  PrintPath(a1,a2,aLength,b1,b2,bLength,path,i,returnVal,pathFP);

  if (lengthNormalize)
   return returnVal/sqrt(aLength*aLength+bLength*bLength);
  else
   return returnVal;
 }
```

Jul 31 17:14 1991 single.c

```
#include <stdio.h>
#include "mylib.h"
#include "types.h"
#include "dict.h"
#include "diff.h"
#include "match.h"
#include "matchparallel.h"

main(argc, argv)
int argc;
char *argv[];
{
 char *dictFile1,*dictFile2,*outFile;
 int shape1,shape2;
 Dictionary dict1,dict2;
 float score;
 char *matchType;
 float centerWeight,normalBandWidth,topToBottom,hillToValleyLocal;
 DiffDescriptorBody dd;
 FILE *pathFP;
 BOOLEAN useL2,slopeConstrain,warp,lengthNormalize,topToBottomOption,hillToValleyOption;
 BOOLEAN separate;

 centerWeight = 1.0;
 normalBandWidth = 20;
 topToBottom = 1.0;
 hillToValleyLocal = 1.0;
 DefArg("%s %d %s %d %s","dict1 shape1 dict2 shape2 outfile",&dictFile1,&shape1,
        &dictFile2,&shape2,&outFile);
 DefOption("-L2","-L2",&useL2);
 DefOption("-slopeConstrain %f","-slopeConstrain <center weight> ",
          &slopeConstrain,&centerWeight);
 DefOption("-warp %f %f","-warp <center weight> <band width> ",
          &warp,&centerWeight,&normalBandWidth);
 DefOption("-separate","-separate",&separate);
 DefOption("-normalize","-normalize",&lengthNormalize);
 DefOption("-topToBottom %f","-topToBottom <ratio>",&topToBottomOption,&topToBottom);
 DefOption("-hillToValley %f","-hillToValley <ratio>",&hillToValleyOption,&hillToValleyLocal);
 ScanArgs(argc,argv);

 dict1 = ReadDictionary(dictFile1);
 dict2 = ReadDictionary(dictFile2);

 if ((shape1 >= dict1->numberOfEntries) || (shape1 < 0) ||
   (shape2 >= dict2->numberOfEntries) || (shape2 < 0))
  DoError("%s: bad shape number.\n",argv[0]);

 if ((pathFP=fopen(outFile,"w"))==NULL)
  DoError("single: error opening output file  %s.\n",outFile);
```

```

  if (useL2) {
   fprintf(stdout,"Using L2 on length normalized shapes.\n");
   dd.diffType = L2;
  }
  else if (slopeConstrain) {
   fprintf(stdout,"Using dynamic time warping with slope contrained to [0.5,2].\n");
   dd.diffType = CONSTRAINED;
   dd.separate = separate;
   if (separate)
    fprintf(stdout,"Top and bottom warped separately.\n");
   else
    fprintf(stdout,"Top and bottom warped together.\n");
  }
  else {
   fprintf(stdout,"Using dynamic time warping with bandwidth %d.\n",normalBandWidth);
   dd.diffType = WARP;
   dd.bandWidth = normalBandWidth;
   dd.separate = separate;
   if (separate)
    fprintf(stdout,"Top and bottom warped separately.\n");
   else
    fprintf(stdout,"Top and bottom warped together.\n");
  }
  if (!useL2) {
   fprintf(stdout,"Center weight = %f.\n",centerWeight);
   dd.centerWeight = centerWeight;
   if (lengthNormalize) {
    dd.lengthNormalize = TRUE;
    fprintf(stdout,"Scores normalized by signal length.\n");
   }
   else
    dd.lengthNormalize = FALSE;
  }
  dd.hillToValley = hillToValleyLocal;
  dd.topToBottom = topToBottom;
  dd.pathFP = pathFP;
  fprintf(stdout,"Top to bottom ratio = %6.2f.\n",topToBottom);
  fprintf(stdout,"Hill to Valley ratio = %6.2f.\n",hillToValleyLocal);

  score = DiffPair(*(dict1->outlines+shape1),
              *(dict2->outlines+shape2),
              &dd);

  fclose(pathFP);

  printf("Score = %f\n",score);
 }
```

Jul 23 20:24 1991 slopeMatch.c

```
float SlopeConstrainedMatch(float *a1,float *a2,int aLength,
                    float *b1,float *b2,int bLength,
                    float maxSlope)
{
 float costs[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH];
 char down[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH];
 char left[MAX_SIGNAL_LENGTH][MAX_SIGNAL_LENGTH];
}
```

Jul 12 14:36 1991 sortMatrix.c

```
#include <stdio.h>
#include "error.h"
#include "pict.h"

#define MAX_ENTRIES 5000

typedef struct {
 float score;
 int x;
 int y;
} *CompareTuple,CompareTupleBody;

int TupleLessThan(CompareTuple *x,CompareTuple *y)
{
 if ((*x)->score == (*y)->score)
  return 0;
 else if ((*x)->score < (*y)->score)
  return -1;
 else
  return 1;
}

void PrintTuple(CompareTuple a,FILE *fp)
{
 fprintf(fp,"(%d,%d): %f\n",a->x,a->y,a->score);
}

void main(int argc,char **argv)
{
 Picture pict;
 int i,j;
 int x,y;
 char *infile;
 CompareTuple scores[MAX_ENTRIES];

 if (argc != 2)
  DoError("Usage: %s infile.\n",argv[0]);
 infile = argv[1];

 pict = load_pict(infile);
 if (pict->width*pict->height > MAX_ENTRIES)
  DoError("%s: matrix has too manyt entries.\n",argv[0]);

 for (y=0,i=0;y<pict->height;++y)
  for (x=0;x<pict->width;++x) {
   CompareTuple temp;
   temp = (CompareTuple)calloc(1,sizeof(CompareTupleBody));
   if (temp==NULL)
       DoError("%s: cannot allocate space.\n",argv[0]);
   temp->score = *((float *)(pict->data)+x+y*pict->width);
   temp->x = x;
   temp->y = y;
```

```
      scores[i] = temp;
      ++i;
     }
    qsort(scores,i,sizeof(CompareTuple),TupleLessThan);
    for (j=0;j<i;++j)
     PrintTuple(scores[j],stdout);
   }
```

Aug 26 17:54 1991 Makefile

```
CCFLAGS = -g -c -I/net/piglet/piglet-1c/hopcroft/new/include

OFUNS = blobify.o orient.o lines.o newBaselines.o newMain.o types.o \
newBlobify.o boxes.o newContour.o numbers.o fontNorm.o \
dict.o

ALPHAOFUNS = orient.o lines.o baselines.o newMain.o types.o \
blobify.o boxes.o newContour.o numbers.o alphaNorm.o \
dict.o


SOURCES = Makefile baselines.c blobify.c boxes.c dict.c dmain.c getAll.c\
getOutline.c lines.c newContour.c newDiff2.c newMain.c\
numbers.c orient.c overlay.c fontNorm.c testFine.c types.c

EXTRNS = /net/piglet/piglet-1c/hopcroft/error/error.o\
/net/piglet/piglet-1c/hopcroft/new/pict/pict.o \
/net/piglet/piglet-1c/hopcroft/lists/lists.o

INCLUDE = /net/piglet/piglet-1c/hopcroft/new/include/
MISC = $(INCLUDE)misc.h
BOOLEAN = $(INCLUDE)boolean.h
LINES = $(INCLUDE)lines.h
LISTS = $(INCLUDE)lists.h
PICT = $(INCLUDE)pict.h
TYPES = $(INCLUDE)types.h
MYLIB = $(INCLUDE)mylib.h
ORIENT = $(INCLUDE)orient.h
BASELINES = $(INCLUDE)baselines.h
BLOBIFY = $(INCLUDE)blobify.h
BOXES = $(INCLUDE)boxes.h
CONTOUR = $(INCLUDE)newContour.h
DIFF = $(INCLUDE)diff.h
DICT = $(INCLUDE)dict.h
ERROR = $(INCLUDE)error.h
FONTNORM = $(INCLUDE)fontNorm.h

orient:     $(OFUNS)
        gcc $(OFUNS) $(HOME)/new/lib/mylib.a /usr/lib/debug/malloc.o -lm -o $@

newBlobify: newBlobify.o
        gcc newBlobify.o ../lib/mylib.a -lm -o $@

makeAlphabet:    $(ALPHAOFUNS)
        gcc $(ALPHAOFUNS) /usr/lib/debug/malloc.o $(EXTRNS) -lm -o $@

overlay:      overlay.o
        gcc overlay.o $(EXTRNS) -o $@

testFine:      testFine.o lines.o guassian.o types.o
        gcc testFine.o lines.o guassian.o types.o $(EXTRNS) -lm -o $@

```

```
boxes:	boxes.o lines.o types.o
	gcc boxes.o lines.o types.o $(HOME)/new/lib/mylib.a -lm -o $@

getOutline:	dict.o getOutline.o
	gcc getOutline.o dict.o $(EXTRNS) -lm -o $@

getAll:	dict.o getAll.o
	gcc getAll.o dict.o $(EXTRNS) -lm -o $@

maxFilter:	maxFilter.o
	gcc maxFilter.o $(HOME)/new/lib/mylib.a -lm -o $@

myWc: myWc.o
	gcc myWc.o $(EXTRNS) -o $@

printCode:	$(SOURCES)
	/usr/5bin/pr -n3 $(SOURCES) | lpr -PWeeklyWorldNews

newBaselines.o:	newBaselines.c $(BOOLEAN) $(PICT) $(TYPES) $(LISTS) $(LINES)\
$(BASELINES)
	gcc $(CCFLAGS) newBaselines.c

blobify.o:	blobify.c $(BOOLEAN) $(PICT) $(BLOBIFY)
	gcc $(CCFLAGS) blobify.c

boxes.o:		boxes.c $(BOOLEAN) $(PICT) $(TYPES) $(BOXES)
	gcc $(CCFLAGS) boxes.c

dict.o:		dict.c $(BOOLEAN) $(TYPES) $(ERROR) $(PICT) $(DICT)
	gcc $(CCFLAGS) dict.c

dmain.o: dmain.c $(BOOLEAN) $(PICT) $(DIFF)
	gcc $(CCFLAGS) dmain.c

getAll.o: getAll.c $(BOOLEAN) $(TYPES) $(PICT) $(DICT)
	gcc $(CCFLAGS) getAll.c

getOutline.o: getOutline.c $(BOOLEAN) $(TYPES) $(PICT) $(DICT)
	gcc $(CCFLAGS) getOutline.c

guassian.o:		guassian.c
	gcc $(CCFLAGS) guassian.c

lines.o:	lines.c $(BOOLEAN) $(PICT) $(LINES)
	gcc $(CCFLAGS) lines.c

maxFilter.o: maxFilter.c $(MYLIB)
	gcc $(CCFLAGS) maxFilter.c

myWc.o:		myWc.c $(BOOLEAN) $(ERROR)
	gcc $(CCFLAGS) myWc.c

newBlobify.o:	newBlobify.c $(MYLIB) $(BLOBIFY)
	gcc $(CCFLAGS) newBlobify.c
```

```
newContour.o:    newContour.c $(BOOLEAN) $(PICT) $(TYPES) $(LINES) \
$(LISTS) $(CONTOUR) $(FONTNORM)
        gcc $(CCFLAGS) newContour.c

newDiff2.o: newDiff2.c $(BOOLEAN) $(TYPES) $(PICT) $(DIFF)
        gcc $(CCFLAGS) newDiff2.c

newMain.o:       newMain.c $(BOOLEAN) $(PICT) $(LISTS) $(LINES) \
$(ORIENT) $(BASELINES) $(BLOBIFY) $(BOXES) $(CONTOUR) $(ORIENT)
        gcc $(CCFLAGS) newMain.c

numbers.o:  numbers.c $(BOOLEAN) $(PICT) $(LINES)
        gcc $(CCFLAGS) numbers.c

orient.o:    orient.c $(BOOLEAN) $(TYPES) $(PICT) $(ORIENT) $(LINES)
        gcc $(CCFLAGS) orient.c

overlay.o:   overlay.c $(BOOLEAN) $(PICT)
        gcc $(CCFLAGS) overlay.c

postproc.o: postproc.c $(BOOLEAN) $(TYPES) $(ERROR) $(PICT) $(DICT)
        gcc $(CCFLAGS) postproc.c

alphaNorm.o: alphaNorm.c $(BOOLEAN) $(TYPES) $(ERROR) $(PICT) $(DICT) $(FONTNORM)
        gcc $(CCFLAGS) alphaNorm.c

fontNorm.o: fontNorm.c $(BOOLEAN) $(TYPES) $(ERROR) $(PICT) $(DICT) $(FONTNORM)
        gcc $(CCFLAGS) fontNorm.c

testFine.o:  testFine.c
        gcc $(CCFLAGS) testFine.c

types.o:              types.c $(TYPES) $(ERROR)
        gcc $(CCFLAGS) types.c
```

Aug 5 15:45 1991 alphaNorm.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"
#include "fontNorm.h"

/* This file is just like fontNorm.c, but assumes that the input is data for an alphabet
dictionary.
 * This data is
 *
 * a-z
 *
 * A-Z
 *
 * 0-9
 *
 * |!@#$%^&*()+\-={}[];:/<>?
 *
 * '`"~
 *
 * _,.
 *
 * The x height will be measured from the x(23). The ascender height will be measured
 * from the l(11).
 */

#define X_HEIGHT_SHAPE 23
#define ASC_HEIGHT_SHAPE 11

extern double ceil(double);
extern int irint(double);


#define UP 0
#define DOWN 1
typedef int Direction;

extern Picture thePict;

void StoreRawOutlinePair(Dictionary dict, int dictEntry,
                         Box box,int *bothX,int *topY, int *baseY,
                         int numberOfLegs)
{
 RawOutlinePair temp;
 int i;
 int *xCursor,*topCursor,*bottomCursor;

 temp = (RawOutlinePair)calloc(1,sizeof(RawOutlinePairBody));
 if (temp == NULL)
```

```
    DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

   temp->box = box;
   temp->numberOfLegs = numberOfLegs;

   temp->x = (int *)calloc(temp->numberOfLegs,sizeof(int));
   temp->top = (int *)calloc(temp->numberOfLegs,sizeof(int));
   temp->bottom = (int *)calloc(temp->numberOfLegs,sizeof(int));
   if ((temp->x == NULL) ||
     (temp->top == NULL) ||
     (temp->bottom == NULL))
    DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

   xCursor = temp->x;
   topCursor = temp->top;
   bottomCursor = temp->bottom;

   for (i=0;i<numberOfLegs;++i) {
    *xCursor++ = *bothX++;
    *topCursor++ = *topY++;
    *bottomCursor++ = *baseY++;
   }
   *(dict->rawOutlines+dictEntry) = temp;
  }

  int RawOutlineWidth(RawOutlinePair a,int middleLine)
  {
   int i,numberOfLegs,right,left;
   int *topCursor,*bottomCursor;
   int topValue,bottomValue;

   numberOfLegs = a->numberOfLegs;

   topCursor = a->top;
   bottomCursor = a->bottom;
   for (i=0;i<numberOfLegs;++i) {
    topValue = *topCursor++;
    bottomValue = *bottomCursor++;

    if (topValue != HIT_THE_BOX) {
     topValue = middleLine - topValue;
     if (topValue<0)
         topValue = 0;
    }
    else
     topValue = 0;

    if (bottomValue != HIT_THE_BOX) {
     bottomValue = bottomValue - middleLine;
     if (bottomValue < 0)
         bottomValue = 0;
    }
    else
     bottomValue = 0;

```

```
    if ((bottomValue != 0)||(topValue != 0))
     break;
   }
   left = i;

   topCursor = a->top+numberOfLegs-1;
   bottomCursor = a->bottom+numberOfLegs-1;
   for (i=numberOfLegs-1;i>=0;--i) {
    topValue = *topCursor--;
    bottomValue = *bottomCursor--;

    if (topValue != HIT_THE_BOX) {
     topValue = middleLine - topValue;
     if (topValue<0)
         topValue = 0;
    }
    else
     topValue = 0;

    if (bottomValue != HIT_THE_BOX) {
     bottomValue = bottomValue - middleLine;
     if (bottomValue < 0)
         bottomValue = 0;
    }
    else bottomValue = 0;

    if ((topValue != 0)||(bottomValue !=0))
     break;
   }
   right = i+1;

   return right-left;
  }

  void ResampleOutlinePair(OutlinePair a,float newToOldFactor)
  /* Resample an outline pair using linear interpolation. */
  {
   int newWidth,oldWidth,i;
   int oldLeft,oldRight;
   float oldCenter;
   float *newX,*newTop,*newBottom;
   float *xCursor,*topCursor,*bottomCursor;

   oldWidth = a->numberOfLegs;
   newWidth = irint(newToOldFactor*oldWidth);

   newX = (float *)calloc(newWidth,sizeof(float));
   newTop = (float *)calloc(newWidth,sizeof(float));
   newBottom = (float *)calloc(newWidth,sizeof(float));
   if ((newX==NULL)||(newTop==NULL)||(newBottom==NULL))
    DoError("ResampleOutlinePair: cannot allocate space.\n",NULL);

   xCursor = newX;
   topCursor = newTop;
   bottomCursor = newBottom;
```

```
 for (i=0;i<newWidth;++i) {
  oldCenter = i/(float)newWidth*(float)oldWidth;
  oldLeft = irint(floor(oldCenter));
  oldRight = irint(ceil(oldCenter));
  if (oldLeft==oldRight) {
   *xCursor++ = *(a->x+oldLeft);
   *topCursor++ = *(a->top+oldLeft);
   *bottomCursor++ = *(a->bottom+oldLeft);
  }
  else {
   float slope;
   slope = *(a->x+oldRight)-*(a->x+oldLeft);
   *xCursor++ = *(a->x+oldLeft) + (oldCenter-oldLeft)*slope;
   slope = *(a->top+oldRight)-*(a->top+oldLeft);
   *topCursor++ = *(a->top+oldLeft) + (oldCenter-oldLeft)*slope;
   slope = *(a->bottom+oldRight)-*(a->bottom+oldLeft);
   *bottomCursor++ = *(a->bottom+oldLeft) + (oldCenter-oldLeft)*slope;
  }
 }

 free(a->x);
 free(a->top);
 free(a->bottom);

 a->x = newX;
 a->top = newTop;
 a->bottom = newBottom;
 a->numberOfLegs = newWidth;
}

void StoreOutlinePair(Dictionary dict, int dictEntry,
                  int middleLine,int fontXHeight,
                  int ascenderHeight)
/* This routine normalizes the raw outline pair stored in dict at dictEntry using the following
 * operations:
 * 1) For the top contour, shift so that the middle line is at y=0 and negate so that the
 *    higher points are greater than 0. For the bottom, shift so that middle line is at y=0,
 *    but don't flip. Thus, lower points have y coordinates greater than 0.
 *    Consider points whose value is HIT_THE_BOX to be at y=0. These correspond to gaps
 *    between the letters.
 * 2) Compress top and bottom y coordinates by 1/fontXHeight so that the coordinates at the
 *    distance of the fontXHeight have value 1. Note that 1 is an arbitrary number. It is
 *    unlikely that a signal will have parts that are the x height above the center line
 *    anyway.
 *    FOR TOP CONTOUR,
 *    IF HEIGHT IS GREATER THAN XHEIGHT, SCALE DIFFERENCE BY 1.5/ASCENDER_HEIGHT.
 *    ELSE SCALE DIFFERENCE BY 1/XHEIGHT.
 *    FOR BOTTOM CONTOUR,
 *    SCALE BY 1.5/ASCENDER_HEIGHT.
 * 3) Compress the x coordinates by the same factor as in step 2. Note that this does not
 *    actually resample the contour. NOW DO THIS WITH RESAMPLE. USE SCALE FACTOR OF
 *    20/XHEIGHT.
 * 4) Remove left and right ends of the contour that have y values of zero. This is so the
```

```
 *   contour starts where the word starts, rather than at the edge of its bouding box.
 * 5) Resample the contour to stretch by firstFontXwidth/fontxWidth.  KILL THIS
OPERATION.
 */
{
 RawOutlinePair raw;
 OutlinePair temp;
 int i,numberOfLegs;
 int y;
 int offset;

 int *xSCursor,*topSCursor,*bottomSCursor;
 float *xDCursor,*topDCursor,*bottomDCursor;
 float *xCursor,*topCursor,*bottomCursor;
 int left,right;
 float foffset;

 raw = *(dict->rawOutlines+dictEntry);

 temp = (OutlinePair)calloc(1,sizeof(OutlinePairBody));
 if (temp == NULL)
  DoError("StoreOutlinePair: cannot allocate space\n",NULL);

 temp->x = (float *)calloc(raw->numberOfLegs,sizeof(float));
 temp->top = (float *)calloc(raw->numberOfLegs,sizeof(float));
 temp->bottom = (float *)calloc(raw->numberOfLegs,sizeof(float));
 if ((temp->x == NULL) ||
   (temp->top == NULL) ||
   (temp->bottom == NULL))
  DoError("StoreOutlinePair: cannot allocate space\n",NULL);

 temp->box = raw->box;
 temp->blackoutHeight = 0;
 temp->numberOfLegs = raw->numberOfLegs;
 offset = temp->offset = *(raw->x);
 temp->width = *(raw->x+raw->numberOfLegs-1) - temp->offset;

 xDCursor = temp->x;
 topDCursor = temp->top;
 bottomDCursor = temp->bottom;
 xSCursor = raw->x;
 topSCursor = raw->top;
 bottomSCursor = raw->bottom;

 numberOfLegs = raw->numberOfLegs;
 for (i=0;i<numberOfLegs; ++i) {
/*   *xDCursor++ = (float)(*xSCursor++ - offset)/fontXHeight; */
  if (*topSCursor==HIT_THE_BOX) {
   y = 0;
   topSCursor++;
  }
  else {
   y = middleLine - *topSCursor++;
   if (y<0)
      y = 0;
```

```
    }
    if (y>fontXHeight/2)
     *topDCursor++ = (float)y *1.5 / ascenderHeight;
    else
     *topDCursor++ = (float)y / fontXHeight;

    if (*bottomSCursor==HIT_THE_BOX) {
     y = 0;
     bottomSCursor++;
    }
    else {
     y = *bottomSCursor++ - middleLine;
     if (y<0)
         y = 0;
    }
    if (y>fontXHeight/2)
     *bottomDCursor++ = (float)y / fontXHeight;
    else
     *bottomDCursor++ = (float)y * 1.5 / ascenderHeight;
   }

  /* Now try to remove parts of the contour on to the left and right of the
   * word shape that are at height 0 */

  /* Find left edge */
   topDCursor = temp->top;
   bottomDCursor = temp->bottom;
   for (i=0;i<numberOfLegs;++i) {
    if ((*topDCursor++ != 0)||(*bottomDCursor++!=0))
     break;
   }
   left = i;

  /* Find right edge */
   topDCursor = temp->top+numberOfLegs-1;
   bottomDCursor = temp->bottom+numberOfLegs-1;
   for (i=numberOfLegs-1;i>=0;--i) {
    if ((*topDCursor-- != 0)||(*bottomDCursor-- !=0))
     break;
   }
   right = i+1;

  /* Clip the ends of the contour at left and right */
   xDCursor = temp->x;
   topDCursor = temp->top;
   bottomDCursor = temp->bottom;
   xCursor = temp->x+left;
   topCursor = temp->top+left;
   bottomCursor = temp->bottom+left;
   foffset = *xSCursor;
   for (i=left;i<right;++i) {
    *xDCursor++ = *xCursor++ - foffset;
    *topDCursor++ = *topCursor++;
    *bottomDCursor++ = *bottomCursor++;
   }
```

```
 temp->numberOfLegs = right-left;

 *(dict->outlines+dictEntry) = temp;
 ResampleOutlinePair(*(dict->outlines+dictEntry),(float)20/(float)fontXHeight);
}

static int lineSpacing;
int OrderOutlinePair(OutlinePair *o1,OutlinePair *o2)
{
 int yDistance;
 int xDistance;
 yDistance = (*o1)->box->pageY - (*o2)->box->pageY;
 if (yDistance<lineSpacing && yDistance > -lineSpacing) {
  xDistance = (*o1)->box->pageX - (*o2)->box->pageX;
  return xDistance;
 }
 return yDistance;
}

void SortDictionary(Dictionary dict)
{
 lineSpacing = 20;
 qsort(dict->rawOutlines,dict->numberOfEntries,sizeof(RawOutlinePair),
       OrderOutlinePair);
}

/* WARNING - assumes at least on entry is not equal to HIT_THE_BOX */
float MaxTopValue(RawOutlinePair o)
{
 int i;
 float maxValue;
 maxValue = *(o->top);
 for (i=0;i<o->numberOfLegs; + + i)
  if (*(o->top+i)>maxValue && (*o->top+i)!=HIT_THE_BOX)
    maxValue = *(o->top+i);
 return maxValue;
}

/* WARNING - assumes at least on entry is not equal to HIT_THE_BOX */
float MinTopValue(RawOutlinePair o)
{
 int i;
 float minValue;
 minValue = *(o->top);
 for (i=0;i<o->numberOfLegs; + + i)
  if (*(o->top+i)<minValue && (*o->top+i)!=HIT_THE_BOX)
    minValue = *(o->top+i);
 return minValue;
}

#define HIST_SIZE 100
void HistogramMax(int *data,int dataLength,int offset,int sign,int *histogram)
{
 int i,bin;
```

```
 if (sign>0) {
  int maxValue;

  maxValue = *data;
  for (i=0;i<dataLength;++i)
   if (data[i]!=HIT_THE_BOX) {
       maxValue = data[i];
       break;
   }
  for (;i<dataLength;++i)
   if (data[i]!=HIT_THE_BOX && data[i]>maxValue)
       maxValue = data[i];
  if (maxValue != HIT_THE_BOX) {
   bin = maxValue-offset;
   if ((bin>=0)&&(bin<HIST_SIZE))
       histogram[bin]++;
  }
 }
 else {
  int minValue;
  minValue = *data;
  for (i=0;i<dataLength;++i)
   if (data[i]!=HIT_THE_BOX) {
       minValue = data[i];
       break;
   }
  for (;i<dataLength;++i)
   if (data[i]!=HIT_THE_BOX && data[i]<minValue)
       minValue = data[i];
  if (minValue != HIT_THE_BOX) {
   bin = minValue-offset;
   if ((bin>=0)&&(bin<HIST_SIZE))
       histogram[bin]++;
  }
 }
}

void Histogram(int *data,int dataLength, int offset, int *histogram)
{
 int i,bin;

 for (i=0;i<dataLength;++i) {
  if (*data != HIT_THE_BOX) {
   bin = *data-offset;
   if ((bin>=0)&&(bin<HIST_SIZE))
       histogram[bin]++;
  }
  data++;
 }
}

int MaxBin(int *histogram)
{
 int i;
 int maxValue;
```

```
 int maxIndex;

 maxValue = histogram[0];
 maxIndex = 0;
 for (i=0;i<HIST_SIZE; + + i)
  if (histogram[i]>maxValue) {
   maxValue = histogram[i];
   maxIndex = i;
  }
 return maxIndex;
}

int MaxBinAbove(int *histogram,int line)
{
 int i;
 int maxValue;
 int maxIndex;
 int top,bottom;

 for (i=0;i<HIST_SIZE; + + i)
  if (histogram[i] != 0)
   break;

 top = i;
 bottom = (line+top)/2;

 maxValue = histogram[top];
 maxIndex = top;
 for (i=top;i< =bottom; + + i)
  if (histogram[i]>maxValue) {
   maxValue = histogram[i];
   maxIndex = i;
  }
 return maxIndex;
}

void DrawTextLines(Picture thePict,Dictionary dict,int topLine,int bottomLine)
{
 int maxLength;
 int halfWidth;
 int x,y;
 float x2,x3,y2,y3;
 float angle;

 angle = (*(dict->rawOutlines))->box->angle;
 maxLength = thePict->width+thePict->height;
 halfWidth = thePict->width / 2;
 x = topLine * -sin(angle) + halfWidth * cos(angle);
 y = topLine * cos(angle) + halfWidth * sin(angle);
 x2 = x+maxLength*cos(angle);
 y2 = y+maxLength*sin(angle);
 x3 = x-maxLength*cos(angle);
 y3 = y-maxLength*sin(angle);
 DrawLine(thePict,x,y,(int)x2,(int)y2,5);
 DrawLine(thePict,x,y,(int)x3,(int)y3,5);
```

```
  x = bottomLine * -sin(angle) + halfWidth * cos(angle);
  y = bottomLine * cos(angle) + halfWidth * sin(angle);
  x2 = x+maxLength*cos(angle);
  y2 = y+maxLength*sin(angle);
  x3 = x-maxLength*cos(angle);
  y3 = y-maxLength*sin(angle);
  DrawLine(thePict,x,y,(int)x2,(int)y2,5);
  DrawLine(thePict,x,y,(int)x3,(int)y3,5);
}

void PageStatistics(Dictionary dict,char *fileName)
/* WARNING - this must be run before PostProcess since PostProcess changes the raw
 * shape data. */
{
  int index;
  int temp;
  int i,startIndex,firstY,minY,endIndex,shape;
  int tops[HIST_SIZE];
  int bottoms[HIST_SIZE];
  int ascenders[HIST_SIZE];
  int descenders[HIST_SIZE];
  int middleLine,topLine,bottomLine,ascenderLine,descenderLine;
  int ascenderHeight,descenderHeight,lineNumber;
  int fontXHeight,fontXWidth,xIndex;
  RawOutlinePair thisShape;
  FILE *fp;
  BOOLEAN haveFirstFontXWidth = FALSE;
  int firstFontXWidth;

  if ((fp=fopen(fileName,"w"))==NULL)
    DoError("PageStatistics: error opening output file %s.\n",fileName);

  SortDictionary(dict);

  index = 0;
  lineNumber = 0;
  while (index < dict->numberOfEntries) {
    startIndex = index;
    firstY = (*(dict->rawOutlines+index))->box->pageY;
    minY = firstY;
    while ( (*(dict->rawOutlines+index))->box->pageY - firstY < 20 &&
            (*(dict->rawOutlines+index))->box->pageY - firstY > -20) {
      if ( minY > ( (*(dict->rawOutlines+index))->box->pageY ))
          minY = (*(dict->rawOutlines+index))->box->pageY;
      ++index;
      if (index == dict->numberOfEntries)
          break;
    }
    endIndex = index;


    /* shapes from start index through endindex are all on */
    /* the same text line */
    /* minY has the top of the highest box on the line. */
```

```
    /* Find the base and toplines by taking the mode of the heights of the
     * valleys of the bottom contours and the peaks of the top contours */
    for (i=0;i<HIST_SIZE;i++) {
     bottoms[i]=0;
    }

    for (shape=startIndex;shape<endIndex;++shape) {
     thisShape = *(dict->rawOutlines+shape);
     Histogram(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);
    }
    bottomLine = MaxBin(bottoms)+minY;
    if (X_HEIGHT_SHAPE>=startIndex&&X_HEIGHT_SHAPE<endIndex) {
     topLine = MinTopValue(*(dict->rawOutlines+X_HEIGHT_SHAPE));
     fontXHeight = bottomLine - topLine;
    }
    if (ASC_HEIGHT_SHAPE>=startIndex&&ASC_HEIGHT_SHAPE<endIndex) {
     ascenderLine = MinTopValue(*(dict->rawOutlines+ASC_HEIGHT_SHAPE));
     ascenderHeight = bottomLine - ascenderLine;
    }
    middleLine = bottomLine-fontXHeight/2;
    topLine = bottomLine-fontXHeight;

    if (thePict)
     DrawTextLines(thePict,dict,topLine,bottomLine);


    fprintf(fp,"%d: %d %d %2.6f\n",lineNumber,fontXHeight,ascenderHeight,
          (float)ascenderHeight/(float)fontXHeight);

    for (shape=startIndex;shape<endIndex;++shape)
     StoreOutlinePair(dict,shape,middleLine,fontXHeight,ascenderHeight);

    ++lineNumber;
   } /* Do another line of text */
   fclose(fp);
  }
```

Aug 21 19:50 1991 baselines.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"
#include "types.h"
#include "lists.h"
#include "lines.h"
#include "baselines.h"

extern double sqrt(double);
extern int irint(double);

/*inline*/ int NewReadPixel(UCHAR *base,int width,float x,float y)
{
 int xi;
 int yi;
 UCHAR mask;

 xi = irint(x);
 yi = irint(y);
 mask = 0x80 >> (xi & 0x7);
 return *(base+yi*width+(xi>>3)) & mask;
}

void NewCountLine1Bit(Picture pict,int x1,int y1,int x2,int y2,int *black,int *blackEdge)
{
 float x,y;
 float xinc,yinc;
 float xupinc,yupinc;
 float den;
 int b,be;
 int width,ucharWidth,height;
 UCHAR *data;

 width = pict->width;
 ucharWidth = pict->uchar_width;
 height = pict->height;
 data = pict->data;

 den = sqrt((y2-y1)*(y2-y1)+(x2-x1)*(x2-x1));
 xinc = (x2-x1)/den;
 yinc = (y2-y1)/den;
 xupinc = -yinc;
 yupinc = xinc;
 x = x1;
 y = y1;

 b=0;
 be=0;

 while (x<width&&x>=0&&y<height&&y>=0) {
```

```
    ++b;
    if (NewReadPixel(data,ucharWidth,x,y)) {
     if (!(NewReadPixel(data,ucharWidth,x+xupinc,y+yupinc) &&
           NewReadPixel(data,ucharWidth,x-xupinc,y-yupinc)))

        ++be;
    }
    x += xinc;
    y += yinc;

  }
  *black = b;
  *blackEdge = be;
}


#define MIN_BLACK 5
float NewCountLine(Picture pict,int x1,int y1,int x2,int y2)
{
  int black,blackEdge;
  black = 0;
  blackEdge = 0;
  NewCountLine1Bit(pict,x1,y1,x2,y2,&black,&blackEdge);
  NewCountLine1Bit(pict,x1,y1,x1-(x2-x1),y1-(y2-y1),&black,&blackEdge);
  if (black < MIN_BLACK)
    return 0;
  else
    return (float)blackEdge/black;
}

static float x2offset;
static float y2offset;
static int projectIndex;
static float *projection;
static int *coordx;
static int *coordy;
BOOLEAN BaseLinePiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
  if (test) {
/*   if (!(projectIndex%10))
      DrawLine(pict,x,y,(int)(x+x2offset),(int)(y+y2offset),0xff); */
/*   WritePixel(pict,x,y,0xff); */
    projection[projectIndex] = NewCountLine(pict,x,y,(int)(x+x2offset),
                                 (int)(y+y2offset));
    coordx[projectIndex] = x;
    coordy[projectIndex++] = y;
    return test;
  } else
    return test;
}

static int lastX;
static int lastY;
BOOLEAN EndPointPiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
```

```
  if (test) {
   lastX = x;
   lastY = y;
  }
  return test;
}

void EndPoints(Picture pict,double angle,int *tx, int *ty,int *bx, int *by)
{
  int xc,yc;
  int maxLength;
  float normal;
  float x2,y2,x3,y3;

  /* Make normal to text point in quadrants I and II */
  /* Assume 0 <= angle < 2*M_PI */
  normal = fmod(angle + M_PI/2,2*M_PI);
  if (normal > M_PI)
   normal -= M_PI;

  xc = pict->width/2;
  yc = pict->height/2;

  maxLength = pict->width+pict->height;
  x2 = xc+maxLength*cos(normal);      /* At bottom of picture */
  y2 = yc+maxLength*sin(normal);
  x3 = xc-maxLength*cos(normal);      /* At top of picture */
  y3 = yc-maxLength*sin(normal);

  LineEngine(pict,xc,yc,(int)x2,(int)y2,0,EndPointPiston);
  *bx = lastX;
  *by = lastY;
  LineEngine(pict,xc,yc,(int)x3,(int)y3,0,EndPointPiston);
  *tx = lastX;
  *ty = lastY;
}

double distance(int x1,int y1,int x2,int y2)
{
  return sqrt((double)((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)));
}

#define BASE_PERCENTILE 0.20
#define MIN_LINE_HEIGHT_FRACTION 0.50
List BaseLines(Picture pict,double angle,char *plotFile)
#ifdef foo
,int *count,
          int **returnCoordx, int **returnCoordy)
#endif
{
  float *topProjection;
  int *topCoordx,*topCoordy;
  int *finalCoordx,*finalCoordy,*finalIndex;
  int topIndex,bottomIndex;
  int topCount,botCount,finalCount;
```

```
int maxLength;
int xc,yc;
float x2,y2,x3,y3;
float maxValue,lastValue;
int i,j;
float baseThresh;
int topX,topY,bottomX,bottomY;
BOOLEAN onTextLine;
List xList,yList,result;
double totalDistance,averageDistance;
FILE *outfile;

printf("angle = %3.3f\n",angle);

maxLength = pict->width+pict->height;

topProjection = (float *)calloc(maxLength,sizeof(float));
topCoordx = (int *)calloc(maxLength,sizeof(int));
topCoordy = (int *)calloc(maxLength,sizeof(int));
finalCoordx = (int *)calloc(maxLength,sizeof(int));
finalCoordy = (int *)calloc(maxLength,sizeof(int));
finalIndex = (int *)calloc(maxLength,sizeof(int));

if ((topProjection == NULL)||
   (topCoordx == NULL)||
   (topCoordy == NULL)||
   (finalIndex == NULL) ||
   (finalCoordx == NULL) ||
   (finalCoordy == NULL)) {
  printf("BaseLines: cannot allocate memory\n");
  exit(-1);
}

EndPoints(pict,angle,&topX,&topY,&bottomX,&bottomY);

printf("Main Line: (%d,%d)-(%d,%d)\n",topX,topY,bottomX,bottomY);
/* DrawLine(pict,topX,topY,bottomX,bottomY,0xff); */

x2offset = maxLength*cos(angle);
y2offset = maxLength*sin(angle);
projectIndex = 0;
projection = topProjection;
coordx = topCoordx;
coordy = topCoordy;
LineEngine(pict,topX,topY,bottomX,bottomY,0,BaseLinePiston);
topCount = projectIndex;

maxValue = topProjection[0];
for (i=0;i<topCount;++i) {
  if (topProjection[i]>maxValue)
    maxValue = topProjection[i];
}

baseThresh = maxValue*BASE_PERCENTILE;
printf("baseThresh = %3.3f\n",baseThresh);
```

```
/* Plot the baseline contour if requested */
 if (plotFile!=NULL) {
  printf("Opening baselines plot file\n");
  if ((outfile = fopen(plotFile,"w"))==NULL) {
   printf("Error opening baseline plot file.\n");
   exit(-1);
  }
  for (i=0;i<topCount;++i)
   fprintf(outfile,"%d %f\n",i,topProjection[i]);
  fprintf(outfile,"\"Projection\n\n");
  fprintf(outfile,
      "0 %f\n%d %f%\n\"Baseline Threshold\n",
        baseThresh,topCount,baseThresh);
 }

 finalCount=0;
 lastValue = topProjection[topCount-1];
 onTextLine = FALSE;
 for (i=1;i<topCount;++i) {
  if (onTextLine) {
   if (lastValue>baseThresh && topProjection[i]<=baseThresh) {
      finalCoordx[finalCount] = topCoordx[i];
      finalCoordy[finalCount] = topCoordy[i];
    finalIndex[finalCount] = i;
      finalCount++;
      onTextLine = FALSE;
   }
  } else {
   if (lastValue<=baseThresh && topProjection[i]>baseThresh) {
      finalCoordx[finalCount] = topCoordx[i];
      finalCoordy[finalCount] = topCoordy[i];
    finalIndex[finalCount] = i;
      finalCount++;
      onTextLine = TRUE;
   }
  }

  lastValue = topProjection[i];
 }
 if (finalCount&1)
  --finalCount;          /* Only take an even number of lines */
 for (totalDistance=0,i=0,j=0;i<finalCount;i+=2) {
  topX = finalCoordx[i];
  topY = finalCoordy[i];
  bottomX = finalCoordx[i+1];
  bottomY = finalCoordy[i+1];
  totalDistance += distance(topX,topY,bottomX,bottomY);
  j+=2;
 }
 averageDistance = totalDistance / (finalCount/2)*MIN_LINE_HEIGHT_FRACTION;
 for (i=0,j=0;i<finalCount;i+=2) {
  topX = finalCoordx[i];
  topY = finalCoordy[i];
  topIndex = finalIndex[i];
```

```
     bottomX = finalCoordx[i+1];
     bottomY = finalCoordy[i+1];
     bottomIndex = finalIndex[i+1];
     finalCoordx[j] = topX;
     finalCoordy[j] = topY;
     finalIndex[j] = topIndex;
     finalCoordx[j+1] = bottomX;
     finalCoordy[j+1] = bottomY;
     finalIndex[j+1] = bottomIndex;
     if (distance(topX,topY,bottomX,bottomY)>averageDistance)
      j+=2;
   }
 #ifdef foo
  *count = j;
  *returnCoordx = finalCoordx;
  *returnCoordy = finalCoordy;
 #endif
  result = nil;
  for (i=j-1;i>=0;--i) {
   push(MakePoint(finalCoordx[i],finalCoordy[i]),result);
  }

  if (plotFile != NULL) {
   fprintf(outfile,"\n0 %f\n",-baseThresh);
   for (i=0;i<j;i+=2) {
        fprintf(outfile,"%d %f\n%d %f\n%d %f\n%d %f\n",
                finalIndex[i],-baseThresh,
                finalIndex[i],-2*baseThresh,
                finalIndex[i+1],-2*baseThresh,
                finalIndex[i+1],-baseThresh);
   }
   fprintf(outfile,"\"Baselines");
   fclose(outfile);
   printf("Done writing baseline plot file.\n");
  }

  return result;
 }

 void DrawBaseLines(Picture pict, List pointList, double angle)
 #ifdef foo
 int count,int *coordx,int *coordy,double angle)
 #endif
 {
  int maxLength;
  float x2,y2,x3,y3;
  int x,y;
  Point temp;
  maxLength = pict->width+pict->height;
  while (!endp(pointList)) {
   temp = pop(pointList);
   x = temp->x;
   y = temp->y;
   x2 = x+maxLength*cos(angle);
   y2 = y+maxLength*sin(angle);
```

```
     x3 = x-maxLength*cos(angle);
     y3 = y-maxLength*sin(angle);
     DrawLine(pict,x,y,(int)x2,(int)y2,0xff);
     DrawLine(pict,x,y,(int)x3,(int)y3,0xff);
   }
  }
```

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"
#include "blobify.h"

static UCHAR bitmasks[] = {0x80,0x40,0x20,0x10,0x8,0x4,0x2,0x1};

Picture Blobify(Picture old,int half_mask_size,double threshold)
{
  Picture new;
  int x,y;
  int tval;
  int left,right,top,bottom;
  int width;
  int *counters;
  int *countptr;
  int mask_size;
  UCHAR *xptr,*xyptr;
  int *leftptr;
  int *rightptr;
  UCHAR *topptr;
  UCHAR *bottomptr;
  int uchar_width;
/* UCHAR bitmask;*/
  int count;
  int inside;
  int thold;
/* Added the following for speedup hack 1/14/91 */
  UCHAR bitMask;
  UCHAR *newCursor;
  UCHAR newValue;
  UCHAR topPixels;
  UCHAR bottomPixels;


  mask_size = 2 * half_mask_size + 1;
/* uchar_width = ROUND8(old->width) >> 3; */
  uchar_width = old->uchar_width;

  left = half_mask_size;
  right = old->width - half_mask_size - 1;

  top = half_mask_size;
  bottom = old->height - half_mask_size -1;


  tval = floor(4*half_mask_size*half_mask_size*threshold);
  new = new_pict(old->width,old->height,old->depth);

  counters = (int *)calloc(old->width,sizeof(int));

```

```
 width = old->width;
 countptr = counters;
 xptr = old->data;
 bitMask = 0x80;
 for (x=0;x<width;++x) {
/*  bitmask = bitmasks[x%8]; */
  xyptr = xptr;
  for (count=0,y=0;y<mask_size;++y) {
   if (*xyptr & bitMask)
        ++count;
   xyptr += uchar_width;
  }
  *(countptr++) = count;
/*  if (x%8 == 7)
 *   ++xptr; */
  if (bitMask == 0x01) {
   bitMask = 0x80;
   ++xptr;
  }
  else
   bitMask = bitMask >> 1;
 }

 for (y=top;y<=bottom;++y) {
  countptr = counters;
  for (inside=0,x=0;x<mask_size;++x)
   inside += *countptr++;

  leftptr = counters;
  rightptr = counters + mask_size;
  newCursor = new->data+y*uchar_width+(left>>3);
  bitMask = bitmasks[left%8];
  newValue = 0;
  for (x=left;x<=right;++x) {
   if (inside>tval)
        /* set pixel */
        newValue |= bitMask;
/*      *(new->data+y*uchar_width+(x>>3)) |= bitmasks[x%8]; */
   if (bitMask == 0x01) {
        bitMask = 0x80;
        *newCursor++ = newValue;
        newValue = 0;
   }
   else
        bitMask = bitMask >> 1;
   inside += *rightptr++;
   inside -= *leftptr++;
  }
  if (bitMask != 0x80) {
   *newCursor = newValue;
  }

  topptr = old->data+(y-half_mask_size)*uchar_width;
  bottomptr = topptr + mask_size*uchar_width;
  countptr = counters;
```

```
    bitMask = 0x01;
    for (x=0;x<width;++x) {
  /*    bitmask = bitmasks[x%8]; */
      if (bitMask == 0x01) {
          topPixels = *topptr++;
          bottomPixels = *bottomptr++;
          bitMask = 0x80;
      }
      else
          bitMask = bitMask >> 1;
      if (topPixels & bitMask) {
          if (!(bottomPixels & bitMask))
           --(*countptr);
      }
      else if (bottomPixels & bitMask)
          ++(*countptr);

      ++countptr;
    }
   }

   return new;
  }

  #ifdef foo
  void main(argc,argv)
  int argc;
  char **argv;
  {
   char *infile,*outfile;
   Picture old,new;
   int half_mask_size;
   float threshold;

   malloc_debug(2);

   if (argc != 5) {
    printf("Usage: %s infile outfile half_mask_size threshold\n",argv[0]);
    exit(0);
   }
   infile = argv[1];
   outfile = argv[2];
   half_mask_size = atoi(argv[3]);
   threshold = atof(argv[4]);

   printf("Loading %s...",infile);
   old = load_pict(infile);
   new = components(old,half_mask_size,threshold);
   write_pict(outfile,new);

  }
  #endif
```

Aug 26 18:10 1991 boxes.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"
#include "types.h"
#include "lists.h"

extern int irint(double);

#define MAX_QUEUE_SIZE 10000
#define BLACK 1
#define WHITE 0

#define ABS(a) ((a)<0?-(a):(a))

typedef Point PointArray;

typedef struct {
 PointBody ulc,lrc;
} MinMaxBox;

typedef struct {
 PointBody xwitness,ywitness;
} WitnessBox;

typedef struct {
 PointArray data;
 int first,last;
 int size;
} QueueBody,*Queue;

Queue MakeQueue(size)
int size;
{
 Queue q;
 if ((q=(Queue)calloc(1,sizeof(QueueBody)))==NULL) {
  printf("Cannot alloc space for queue body\n");
  exit(0);
 }
 if ((q->data=(PointArray)calloc(size,sizeof(PointBody)))==NULL) {
  printf("Cannot allocate space for queue array\n");
  exit(0);
 }
 q->first=q->last=0;
 q->size=size;
 return q;
}

void InsertPoint(x,y,q)
int x,y;
Queue q;
```

```
{
 q->data[q->last].x=x;
 q->data[q->last].y=y;
 q->last=(q->last+1)%q->size;
 if (q->last==q->first) {
  printf("Maximum q size exceeded\n");
  exit(0);
 }
}

void GetFirst(x,y,q)
int *x,*y;
Queue q;
{
 if (q->first==q->last) {
  printf("Error: tried top pop empty queue\n");
  exit(0);
 }
 *x=q->data[q->first].x;
 *y=q->data[q->first].y;
 q->first=(q->first+1)%q->size;
}

BOOLEAN Empty(q)
Queue q;
{
 return q->first==q->last;
}

void InsertBlackNeighbors(Picture pict,int x,int y,Queue queue)
{
 if (ReadPixel(pict,x+1,y)) {
  WritePixel(pict,x+1,y,WHITE);
  InsertPoint(x+1,y,queue);
 }
 if (ReadPixel(pict,x-1,y)) {
  WritePixel(pict,x-1,y,WHITE);
  InsertPoint(x-1,y,queue);
 }
 if (ReadPixel(pict,x,y+1)) {
  WritePixel(pict,x,y+1,WHITE);
  InsertPoint(x,y+1,queue);
 }
 if (ReadPixel(pict,x,y-1)) {
  WritePixel(pict,x,y-1,WHITE);
  InsertPoint(x,y-1,queue);
 }
}

void PointFromTheta(theta,x,y)
float theta;
float *x,*y;
{
 *x = cos(theta);
 *y = sin(theta);
```

```
}

void Normal(x,y,nx,ny)
float x,y;
float *nx,*ny;
{
 *nx = -y;
 *ny = x;
}

int DotFl(fx,fy,ix,iy)
float fx,fy;
int ix,iy;
{
 return irint(fx*ix+fy*iy);
}

static float pox,poy,pnx,pny;

void MinMax(boundingBox,oldFrameBox,px,py)
MinMaxBox *boundingBox;
WitnessBox *oldFrameBox;
int px,py;
{
/* IGNORE THETA FOR THE TIME BEING */
 if (boundingBox->lrc.x < DotFl(pox,poy,px,py)) {
  boundingBox->lrc.x = DotFl(pox,poy,px,py);
 }
 if (boundingBox->lrc.y < DotFl(pnx,pny,px,py)) {
  boundingBox->lrc.y = DotFl(pnx,pny,px,py);
 }
 if (boundingBox->ulc.x > DotFl(pox,poy,px,py)) {
  boundingBox->ulc.x = DotFl(pox,poy,px,py);
  oldFrameBox->xwitness.x = px;
  oldFrameBox->xwitness.y = py;
 }
 if (boundingBox->ulc.y > DotFl(pnx,pny,px,py)) {
  boundingBox->ulc.y = DotFl(pnx,pny,px,py);
  oldFrameBox->ywitness.x = px;
  oldFrameBox->ywitness.y = py;
 }
}

/* Set the pixels on the border of the image to the color WHITE so that
 * the paint routine need never worry about going off the edge of the
 * image. */
void FramePicture(pict)
Picture pict;
{
 int i;
 for (i=0;i<pict->height; ++i) {
  WritePixel(pict,0,i,WHITE);
  WritePixel(pict,pict->width-1,i,WHITE);
 }
 for (i=0;i<pict->width; ++i) {
```

```
   WritePixel(pict,i,0,WHITE);
   WritePixel(pict,i,pict->height-1,WHITE);
  }
 }

/*
 * Given as input a thresholded image, find the borders of the connected
 * components. Assumes image is thresholded to 0 and 1.
 */
void PaintComponent(pict,x,y,queue,boundingBox,oldFrameBox)
Picture pict;
int x,y;
Queue queue;
MinMaxBox *boundingBox;
WitnessBox *oldFrameBox;
{
 boundingBox->ulc.x = boundingBox->lrc.x = DotFl(pox,poy,x,y);
 boundingBox->ulc.y = boundingBox->lrc.y = DotFl(pnx,pny,x,y);
 oldFrameBox->xwitness.x = oldFrameBox->ywitness.x = x;
 oldFrameBox->xwitness.y = oldFrameBox->ywitness.y = y;

 InsertPoint(x,y,queue);
 WritePixel(pict,x,y,WHITE);
/* printf("Queue status: %s\n",(Empty(queue))?"empty":"not empty"); */
 while (!Empty(queue)) {
  GetFirst(&x,&y,queue);
  MinMax(boundingBox,oldFrameBox,x,y);
  InsertBlackNeighbors(pict,x,y,queue);
 }
}

int iabs(int x)
{
 if (x<0)
  return -x;
 else
  return x;
}

BOOLEAN PointInBounds(Picture pict,int x,int y)
{
 return x>=0 && x<pict->width && y>=0 && y<pict->height;
}


BOOLEAN BoxInBounds(Picture pict,int x, int y, int width, int height,
               double angle)
{
 int rightX,rightY,downX,downY;
 rightX = width*cos(angle);
 rightY = width*sin(angle);
 downX = height*cos(angle+M_PI/2);
 downY = height*sin(angle+M_PI/2);
 return (PointInBounds(pict,x,y) &&
       PointInBounds(pict,x+rightX,y+rightY) &&
```

```
            PointInBounds(pict,x+rightX+downX,y+rightY+downY) &&
            PointInBounds(pict,x+downX,y+downY));
}

void GetCorner(WitnessBox *box,int *ulcx,int *ulcy)
{
 double c2;
 c2 = (-pny*(box->ywitness.x-box->xwitness.x) +
     pnx*(box->ywitness.y-box->xwitness.y) ) /
    (pox*pny - pnx*poy);
 *ulcx = c2*pox+box->ywitness.x;
 *ulcy = c2*poy+box->ywitness.y;
}

List FindBorders(Picture pict,double theta)
{
 int x,y;
 int ulcx,ulcy;
 Queue queue;
 MinMaxBox boundingBox;
 WitnessBox oldFrameBox;
 List boxList;
 int width,height;

 queue = MakeQueue(MAX_QUEUE_SIZE);

 PointFromTheta(theta,&pox,&poy);
 Normal(pox,poy,&pnx,&pny);

 printf("Framing picture\n");
 FramePicture(pict);            /* Put a "visited" color border
                                 * around the image */
 boxList = nil;
 for (y=1;y<pict->height-1;++y)
  for (x=1;x<pict->width-1;++x)
   if (ReadPixel(pict,x,y)) {
/*      printf("Found component at (%d,%d)\n",x,y); */
        PaintComponent(pict,x,y,queue,&boundingBox,&oldFrameBox);
/*      printf("Making box: %d %d %d %d\n",
            oldFrameBox.ulc.x,
            oldFrameBox.ulc.y,
            oldFrameBox.lrc.x,
            oldFrameBox.lrc.y);
*/
        GetCorner(&oldFrameBox,&ulcx,&ulcy);
        width = boundingBox.lrc.x-boundingBox.ulc.x;
        height = boundingBox.lrc.y-boundingBox.ulc.y;
/*      if (iabs(height)>10) */
         if (BoxInBounds(pict,ulcx,ulcy,
                       width,height,theta))
           push(MakeBox(ulcx,ulcy,
                       width,height,theta),
                boxList);
   }
 printf("Found %d boxes completely on the page\n",ListLength(boxList));
```

```
  return boxList;
 }

 void DrawBox(Picture pict,Box box)
 {
  int rightX,rightY,downX,downY;
  rightX = box->width*cos(box->angle);
  rightY = box->width*sin(box->angle);
  downX = box->height*cos(box->angle+M_PI/2);
  downY = box->height*sin(box->angle+M_PI/2);
/* printf("DrawBox: %d %d %d %d\n",box->x,box->y,box->width<,box->height); */
  DrawLine(pict,box->x,box->y,box->x+rightX,box->y+rightY,0xff);
  DrawLine(pict,box->x+rightX,box->y+rightY,
          box->x+rightX+downX,box->y+rightY+downY,0xff);
  DrawLine(pict,box->x+rightX+downX,box->y+rightY+downY,
          box->x+downX,box->y+downY,0xff);
  DrawLine(pict,box->x+downX,box->y+downY,box->x,box->y,0xff);
 }

 void DrawColorBox(Picture pict,Box box,int color)
 {
  int rightX,rightY,downX,downY;
  rightX = box->width*cos(box->angle);
  rightY = box->width*sin(box->angle);
  downX = box->height*cos(box->angle+M_PI/2);
  downY = box->height*sin(box->angle+M_PI/2);
/* printf("DrawBox: %d %d %d %d\n",box->x,box->y,box->width<,box->height); */
  DrawLine(pict,box->x,box->y,box->x+rightX,box->y+rightY,color);
  DrawLine(pict,box->x+rightX,box->y+rightY,
          box->x+rightX+downX,box->y+rightY+downY,color);
  DrawLine(pict,box->x+rightX+downX,box->y+rightY+downY,
          box->x+downX,box->y+downY,color);
  DrawLine(pict,box->x+downX,box->y+downY,box->x,box->y,color);
 }


 void DrawBoxList(Picture pict,List boxList)
 {
  while (!endp(boxList)) {
   DrawBox(pict,(Box)pop(boxList));
  }
 }


 #ifdef TRYMAIN
 /* WARNING - be sure to replace the height check in FindBorders */
 #endif
 void main(argc,argv)
 int argc;
 char **argv;
 {
  char *infileName,*outfileName;
  List boxList;
  int width,height;
  float theta;
```

```
 Picture pict,finalPict;
 FILE *outfile;

 if (argc != 4) {
  printf("Usage: %s infile outfile page_orientation\n",argv[0]);
  exit(0);
 }
 infileName = argv[1];
 outfileName = argv[2];
 theta = atof(argv[3]);

 printf("Loading %s...",infileName);
 pict = load_pict(infileName);

 printf("\nFinding boxes.\n");

 finalPict = new_pict(pict->width,pict->height,pict->depth);
/* CopyPicture(finalPict,pict); */
 boxList = FindBorders(pict,theta);

 DrawBoxList(finalPict,boxList);
 write_pict(outfileName,finalPict);
}
```

Jan 16 15:52 1991 dict.c

```
#include <stdio.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"

void WriteOutlinePair(OutlinePair o, FILE *fp)
{
 fwrite(o->box,sizeof(BoxBody),1,fp);
 fwrite(&(o->blackoutHeight),sizeof(float),1,fp);
 fwrite(&(o->numberOfLegs),sizeof(int),1,fp);
 fwrite(&(o->offset),sizeof(int),1,fp);
 fwrite(&(o->width),sizeof(int),1,fp);

 fwrite(o->x,sizeof(float),o->numberOfLegs,fp);
 fwrite(o->top,sizeof(float),o->numberOfLegs,fp);
 fwrite(o->bottom,sizeof(float),o->numberOfLegs,fp);
}

void WriteDictionary(Dictionary dict, char *filename)
{
 FILE *fp;
 int temp;
 int i;
 if ((fp=fopen(filename,"w"))==NULL)
  DoError("WriteDictionary: Error opening output file.\n",NULL);
 temp = 1234567;
 fwrite(&temp,sizeof(int),1,fp);
 fwrite(&(dict->numberOfEntries),sizeof(int),1,fp);

 if (dict->infoString == NULL) {
  temp = 0;
  fwrite(&temp,sizeof(int),1,fp);
 }
 else{
  temp = strlen(dict->infoString)+1;
  fwrite(&temp,sizeof(int),1,fp);
  fwrite(dict->infoString,sizeof(char),temp,fp);
 }

 for (i=0;i<dict->numberOfEntries;++i)
  WriteOutlinePair(*(dict->outlines+i),fp);
 fclose(fp);
}


/* Reads a Box from a binary stream. the type Box is defined in box.h */
Box ReadBox(FILE *fp)
{
 Box temp;
 temp = (Box)calloc(1,sizeof(BoxBody));
```

```
  if (temp==NULL)
   DoError("ReadBox: cannot allocate space\n",NULL);
  if (fread(temp,sizeof(BoxBody),1,fp)!=1)
   DoError("ReadBox: error reading bounding box\n",NULL);
  return temp;
 }

 /* Reads an OutlinePair from a binary stream. The format of an OutlinePair
  * follows:
  *  BoxBody - shape bounding box
  *  float  - blackout bar height
  *  int    - number of legs in the contour
  *  int    - x coordinate of left edge of contour
  *  int    - width in pixels of edge contour
  *  float[numberOfLegs] - x coordinates of contours
  *  float[numberOfLegs] - y coordinates of top contour
  *  float[numberOfLegs] - y coordinates of bototm contour
  */
 OutlinePair ReadOutlinePair(FILE *fp)
 {
  OutlinePair temp;
  temp = (OutlinePair)calloc(1,sizeof(OutlinePairBody));
  if (temp == NULL)
   DoError("ReadOutlinePair: cannot allocate space\n",NULL);
  temp->box = ReadBox(fp);

  if (fread(&(temp->blackoutHeight),sizeof(float),1,fp)!=1)
   DoError("ReadOutlinePair: error reading blackoutHeight\n",NULL);

  if (fread(&(temp->numberOfLegs),sizeof(int),1,fp)!=1)
   DoError("ReadOutlinePair: error reading length\n",NULL);

  if (fread(&(temp->offset),sizeof(int),1,fp)!=1)
   DoError("ReadOutlinePair: error reading offset\n",NULL);
  if (fread(&(temp->width),sizeof(int),1,fp)!=1)
   DoError("ReadOutlinePair: error reading width\n",NULL);

  temp->x = (float *)calloc(temp->numberOfLegs,sizeof(float));
  if (temp->x == NULL)
   DoError("ReadOutlinePair: cannot allocate space\n",NULL);
  if (fread(temp->x,
        sizeof(float),temp->numberOfLegs,fp)!=temp->numberOfLegs)
   DoError("ReadOutlinePair: error reading x coords\n",NULL);

  temp->top = (float *)calloc(temp->numberOfLegs,sizeof(float));
  if (temp->top == NULL)
   DoError("ReadOutlinePair: cannot allocate space\n",NULL);
  if (fread(temp->top,sizeof(float),
        temp->numberOfLegs,fp)!=temp->numberOfLegs)
   DoError("ReadOutlinePair: error reading topY coords\n",NULL);

  temp->bottom = (float *)calloc(temp->numberOfLegs,sizeof(float));
  if (temp->bottom == NULL)
   DoError("ReadOutlinePair: cannot allocate space\n",NULL);
  if (fread(temp->bottom,
```

```
        sizeof(float),temp->numberOfLegs,fp)!=temp->numberOfLegs)
  DoError("ReadOutlinePair: error reading bottomY coords\n",NULL);

 return temp;
}

/* Create a new Dictionary structure with space allocated for the
 * entries. */
Dictionary NewDict(int numberOfEntries)
{
 Dictionary temp;
 temp = (Dictionary)calloc(1,sizeof(DictionaryBody));
 if (temp == NULL)
  DoError("NewDict: cannot allocate space\n",NULL);
 temp->numberOfEntries = numberOfEntries;
 temp->infoString = NULL;
 temp->rawOutlines = (RawOutlinePair *)calloc(numberOfEntries,
                              sizeof(RawOutlinePair));
 temp->outlines = (OutlinePair *)calloc(numberOfEntries,
                              sizeof(OutlinePair));
 if ((temp->outlines == NULL)||(temp->rawOutlines == NULL))
  DoError("NewDict: cannot allocate space\n",NULL);
 return temp;
}

/* Read a dictionary from a binary format file. The file organization
 * follows:
 *  int     - number of entries in the dictionary
 *  OutlinePair[numberOfEntries] - outlines of each shape in the dictionary
 * When a dictionary is read in, the shapes are sorted such that they fall
 * in the order of words on textlines. */
Dictionary ReadDictionary(char *filename)
{
 FILE *fp;
 Dictionary dict;
 int i;
 int temp;
 int infoStringLength;
 int numberOfEntries;
 int magicNumber;

 if ((fp=fopen(filename,"r"))==NULL)
  DoError("Error opening input file\n",NULL);

 if (fread(&magicNumber,sizeof(int),1,fp)!=1)
  DoError("Error reading dictionary\n",NULL);
 if (magicNumber != 1234567)
  DoError("ReadDictionary: input file %s is not a dictionary file.\n",
        filename);

 if (fread(&numberOfEntries,sizeof(int),1,fp)!= 1)
  DoError("Error reading dictionary\n",NULL);
 dict = NewDict(numberOfEntries);

 if (fread(&infoStringLength,sizeof(int),1,fp)!= 1)
```

```
    DoError("Error reading dictionary\n",NULL);
   if (infoStringLength) {
    if ((dict->infoString = (char *)calloc(infoStringLength,sizeof(char))) ==
         NULL)
     DoError("ReadDictionary: cannot allocate space for info string.\n",NULL);
    fread(dict->infoString,infoStringLength,sizeof(char),fp);
    *(dict->infoString+infoStringLength-1) = '\0'; /* Set last char to 0 just in case */
   }

   for (i=0;i<numberOfEntries; ++i)
    *(dict->outlines+i) = ReadOutlinePair(fp);
   fclose(fp);
   return dict;
  }

  char *ArgListToString(int argc, char **argv)
  {
   int i;
   int totalLength;
   char *theString;
   char *destCursor,*srcCursor;

   for (i=0,totalLength=0;i<argc; ++i)
    totalLength += strlen(argv[i]) + 1; /* Room for each arg and one space */
   totalLength++;              /* Room for thee EOS character */

   if ((theString = (char *)calloc(totalLength,sizeof(char)))==NULL)
    DoError("ArgListToString: cannot allocatee space.\n",NULL);

   for (i=0,destCursor=theString;i<argc; ++i) {
    srcCursor = argv[i];
    while (*srcCursor != '\0')
     *destCursor++ = *srcCursor++;
    *destCursor++ = ' ';
   }
   *destCursor = '\0';

   return theString;
  }
```

Jan 11 17:06 1991 dmain.c

```
#include <stdio.h>
#include <math.h>
#include <values.h>
#include "boolean.h"
#include "types.h"
#include "pict.h"
#include "diff.h"


void main(int argc,char **argv)
{
 Picture pict;
 char *infile1,*infile2,*outfile;

 if (argc != 4) {
  printf("Usage:\n");
  printf("  %s infile1 infile2 outfile\n",argv[0]);
  exit(-1);
 }

 infile1 = argv[1];
 infile2 = argv[2];
 outfile = argv[3];
 pict = CompareDictionaries(infile1,infile2);
 WritePictureAsAscii(pict,outfile);
}
```

Jun 21 15:54 1991 fft.c

```
/* Copyright 1991 by Michael Hopcroft.
 * Right is hearby granted to Xerox Corporation to make use of this
 * code free of charge. */
#include <stdio.h>
#include <math.h>
#include "fft.h"

/* Applies bit reversal permutation matrix to array a. length must be a power
 * of 2. */
void BitReverse(float *a, int n)
{
 int i,j,k;
 float temp;

 j=1;
 for (i=1;i<n; ++i) {
  if (i<j) {
   temp = a[i-1];
   a[i-1] = a[j-1];
   a[j-1] = temp;
  }
  k=n/2;
  while (k<j) {
   j = j-k;
   k = k/2;
  }
  j = j+k;
 }
}

#define TWOPI (M_PI*2)

void fft(float *real,float *imag,int logn,int mode)
{
 int n;
 int j,top,i,id,bottom;
 int stage,subpartLength;
 float tempr,tempi,temp2r,temp2i,ar,ai,wr,wi,angle;

 n = irint(exp2((double)logn));

 for (stage=1, subpartLength = n;
    stage<=logn;
    ++stage, subpartLength/=2) {
  angle = TWOPI/subpartLength;
  ar = 1.0;
  ai = 0.0;
  if (mode == REVERSE) {
   wr = cos(angle);
   wi = sin(angle);
  } else {
   wr = cos(angle);
```

```
      wi = -sin(angle);
     }
     for (j=0;j<subpartLength/2;++j) {      /* for each offset in a part */
      for (top=j;top<n;top+=subpartLength) {  /* for each part */
          bottom = top+subpartLength/2;
          tempr = real[bottom];          /* temp = x[id] */
          tempi = imag[bottom];
          real[bottom] = real[top]-real[bottom]; /* x[id] = x[i] - x[id] */
          imag[bottom] = imag[top]-imag[bottom];
          temp2r = real[bottom]*ar-imag[bottom]*ai; /* temp2 = x[id]*a */
          temp2i = real[bottom]*ai+imag[bottom]*ar;
          real[bottom] = temp2r;          /* x[id] = temp2 */
          imag[bottom] = temp2i;
          real[top] += tempr;            /* x[i] +=temp */
          imag[top] += tempi;
      }
      temp2r = ar*wr-ai*wi;          /* a *= w */
      temp2i = ai*wr+ar*wi;
      ar = temp2r;
      ai = temp2i;
     }
    }
    BitReverse(real,n);
    BitReverse(imag,n);

   #ifdef foo
    if (mode == MAGNITUDE)
     for (i=0;i<n;++i)
      real[i] = sqrt(real[i]*real[i]+imag[i]*imag[i]);
   #endif

    if (mode == MAGNITUDE)
     for (i=0;i<n;++i)
      real[i] = sqrt(real[i]*real[i]+imag[i]*imag[i]);
   }

   #ifdef TRYMAIN
   void main(int argc,char **argv)
   {
   #define POWER 8
   #define LENGTH 256
    float real[LENGTH];
    float imag[LENGTH];
    int i;
   #ifdef foo
    for (i=0;i<LENGTH;++i){
     if (i<LENGTH/2)
      real[i] = 1.0;
     else
      real[i] = 0.0;
     imag[i] = 0.0;
    }
   #endif

    for (i=0;i<LENGTH;++i) {
```

```
   real[i] = sin(8*TWOPI*i/(LENGTH-1));
   imag[i] = 0.0;
  }
  fft(real,imag,POWER,MAGNITUDE);
  for (i=0;i<LENGTH; ++i)
   printf("%d %f\n",i,real[i]);
 }
 #endif
```

Aug 15 21:19 1991 fontNorm.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"
#include "fontNorm.h"


extern double ceil(double);
extern int irint(double);


#define UP 0
#define DOWN 1
typedef int Direction;

extern Picture thePict;

void StoreRawOutlinePair(Dictionary dict, int dictEntry,
                        Box box,int *bothX,int *topY, int *baseY,
                        int numberOfLegs)
{
 RawOutlinePair temp;
 int i;
 int *xCursor,*topCursor,*bottomCursor;

 temp = (RawOutlinePair)calloc(1,sizeof(RawOutlinePairBody));
 if (temp == NULL)
  DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

 temp->box = box;
 temp->numberOfLegs = numberOfLegs;

 temp->x = (int *)calloc(temp->numberOfLegs,sizeof(int));
 temp->top = (int *)calloc(temp->numberOfLegs,sizeof(int));
 temp->bottom = (int *)calloc(temp->numberOfLegs,sizeof(int));
 if ((temp->x == NULL) ||
   (temp->top == NULL) ||
   (temp->bottom == NULL))
  DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

 xCursor = temp->x;
 topCursor = temp->top;
 bottomCursor = temp->bottom;

 for (i=0;i<numberOfLegs;++i) {
  *xCursor++ = *bothX++;
  *topCursor++ = *topY++;
  *bottomCursor++ = *baseY++;
 }
```

```
 *(dict->rawOutlines+dictEntry) = temp;
}

int RawOutlineWidth(RawOutlinePair a,int middleLine)
{
 int i,numberOfLegs,right,left;
 int *topCursor,*bottomCursor;
 int topValue,bottomValue;

 numberOfLegs = a->numberOfLegs;

 topCursor = a->top;
 bottomCursor = a->bottom;
 for (i=0;i<numberOfLegs;++i) {
  topValue = *topCursor++;
  bottomValue = *bottomCursor++;

  if (topValue != HIT_THE_BOX) {
   topValue = middleLine - topValue;
   if (topValue<0)
       topValue = 0;
  }
  else
   topValue = 0;

  if (bottomValue != HIT_THE_BOX) {
   bottomValue = bottomValue - middleLine;
   if (bottomValue < 0)
       bottomValue = 0;
  }
  else
   bottomValue = 0;

  if ((bottomValue != 0)||(topValue != 0))
   break;
 }
 left = i;

 topCursor = a->top+numberOfLegs-1;
 bottomCursor = a->bottom+numberOfLegs-1;
 for (i=numberOfLegs-1;i>=0;--i) {
  topValue = *topCursor--;
  bottomValue = *bottomCursor--;

  if (topValue != HIT_THE_BOX) {
   topValue = middleLine - topValue;
   if (topValue<0)
       topValue = 0;
  }
  else
   topValue = 0;

  if (bottomValue != HIT_THE_BOX) {
   bottomValue = bottomValue - middleLine;
   if (bottomValue < 0)
```

```
        bottomValue = 0;
    }
    else bottomValue = 0;

    if ((topValue != 0)||(bottomValue !=0))
      break;
  }
  right = i+1;

  return right-left;
}

void ResampleOutlinePair(OutlinePair a,float newToOldFactor)
/* Resample an outline pair using linear interpolation. */
{
  int newWidth,oldWidth,i;
  int oldLeft,oldRight;
  float oldCenter;
  float *newX,*newTop,*newBottom;
  float *xCursor,*topCursor,*bottomCursor;

  oldWidth = a->numberOfLegs;
  newWidth = irint(newToOldFactor*oldWidth);

  newX = (float *)calloc(newWidth,sizeof(float));
  newTop = (float *)calloc(newWidth,sizeof(float));
  newBottom = (float *)calloc(newWidth,sizeof(float));
  if ((newX==NULL)||(newTop==NULL)||(newBottom==NULL))
    DoError("ResampleOutlinePair: cannot allocate space.\n",NULL);

  xCursor = newX;
  topCursor = newTop;
  bottomCursor = newBottom;

  for (i=0;i<newWidth;++i) {
    oldCenter = i/(float)newWidth*(float)oldWidth;
    oldLeft = irint(floor(oldCenter));
    oldRight = irint(ceil(oldCenter));
    if (oldLeft==oldRight) {
      *xCursor++ = *(a->x+oldLeft);
      *topCursor++ = *(a->top+oldLeft);
      *bottomCursor++ = *(a->bottom+oldLeft);
    }
    else {
      float slope;
      slope = *(a->x+oldRight)-*(a->x+oldLeft);
      *xCursor++ = *(a->x+oldLeft) + (oldCenter-oldLeft)*slope;
      slope = *(a->top+oldRight)-*(a->top+oldLeft);
      *topCursor++ = *(a->top+oldLeft) + (oldCenter-oldLeft)*slope;
      slope = *(a->bottom+oldRight)-*(a->bottom+oldLeft);
      *bottomCursor++ = *(a->bottom+oldLeft) + (oldCenter-oldLeft)*slope;
    }
  }

  free(a->x);
```

```
 free(a->top);
 free(a->bottom);

 a->x = newX;
 a->top = newTop;
 a->bottom = newBottom;
 a->numberOfLegs = newWidth;
}

void StoreOutlinePair(Dictionary dict, int dictEntry,
               int middleLine,int fontXHeight,
               int ascenderHeight,NormalizationDescriptor *nd)
/* This routine normalizes the raw outline pair stored in dict at dictEntry using the following
 * operations:
 * 1) For the top contour, shift so that the middle line is at y=0 and negate so that the
 *   higher points are greater than 0. For the bottom, shift so that middle line is at y=0,
 *   but don't flip. Thus, lower points have y coordinates greater than 0.
 *   Consider points whose value is HIT_THE_BOX to be at y=0. These correspond to gaps
 *   between the letters.
 * 2) Compress top and bottom y coordinates by 1/fontXHeight so that the coordinates at
the
 *   distance of the fontXHeight have value 1. Note that 1 is an arbitrary number. It is
 *   unlikely that a signal will have parts that are the x height above the center line
 *   anyway.
 *   FOR TOP CONTOUR,
 *   IF HEIGHT IS GREATER THAN XHEIGHT, SCALE DIFFERENCE BY 1.5/ASCENDER_HEIGHT.
 *   ELSE SCALE DIFFERENCE BY 1/XHEIGHT.
 *   FOR BOTTOM CONTOUR,
 *   SCALE BY 1.5/ASCENDER_HEIGHT.
 * 3) Compress the x coordinates by the same factor as in step 2. Note that this does not
 *   actually resample the contour. NOW DO THIS WITH RESAMPLE. USE SCALE FACTOR OF
 *   20/XHEIGHT.
 * 4) Remove left and right ends of the contour that have y values of zero. This is so the
 *   contour starts where the word starts, rather than at the edge of its bouding box.
 * 5) Resample the contour to stretch by firstFontXwidth/fontxWidth. KILL THIS
OPERATION.
 */
{
 RawOutlinePair raw;
 OutlinePair temp;
 int i,numberOfLegs;
 int y;
 int offset;
 int *xSCursor,*topSCursor,*bottomSCursor;
 float *xDCursor,*topDCursor,*bottomDCursor;
 float *xCursor,*topCursor,*bottomCursor;
 int left,right;
 float foffset;
 float ascenderFactor,xHeightFactor,widthFactor;

 raw = *(dict->rawOutlines+dictEntry);

 temp = (OutlinePair)calloc(1,sizeof(OutlinePairBody));
 if (temp == NULL)
  DoError("StoreOutlinePair: cannot allocate space\n",NULL);
```

```
    temp->x = (float *)calloc(raw->numberOfLegs,sizeof(float));
    temp->top = (float *)calloc(raw->numberOfLegs,sizeof(float));
    temp->bottom = (float *)calloc(raw->numberOfLegs,sizeof(float));
    if ((temp->x == NULL) ||
      (temp->top == NULL) ||
      (temp->bottom == NULL))
     DoError("StoreOutlinePair: cannot allocate space\n",NULL);

    temp->box = raw->box;
    temp->blackoutHeight = 0;
    temp->numberOfLegs = raw->numberOfLegs;
    offset = temp->offset = *(raw->x);
    temp->width = *(raw->x+raw->numberOfLegs-1) - temp->offset;

    xDCursor = temp->x;
    topDCursor = temp->top;
    bottomDCursor = temp->bottom;
    xSCursor = raw->x;
    topSCursor = raw->top;
    bottomSCursor = raw->bottom;

    ascenderFactor = 1.5/ascenderHeight;
    xHeightFactor = 1.0/fontXHeight;
    widthFactor = 20.0/fontXHeight;
    if (nd->noXHeightNormalize) {
     xHeightFactor = 1.0;
     ascenderFactor = 1.0;
    }
    if (nd->noAscenderNormalize)
     ascenderFactor = xHeightFactor;

    numberOfLegs = raw->numberOfLegs;
    for (i=0;i<numberOfLegs; ++i) {
     if (*topSCursor==HIT_THE_BOX) {
      y = 0;
      topSCursor++;
     }
     else {
      y = middleLine - *topSCursor++;
      if (y<0)
          y = 0;
     }
     if (y>fontXHeight/2) {
      float temp1 = (float)y * ascenderFactor;
      float temp2 = (float)fontXHeight/2 * xHeightFactor;
      if (temp1<temp2)
          *topDCursor++ = temp2;
      else
          *topDCursor++ = temp1;
  /*
      *topDCursor++ = (float)y * ascenderFactor;
  */
     }
     else
```

```
      *topDCursor++ = (float)y * xHeightFactor;

     if (*bottomSCursor==HIT_THE_BOX) {
      y = 0;
      bottomSCursor++;
     }
     else {
      y = *bottomSCursor++ - middleLine;
      if (y<0)
          y = 0;
     }
     if (y<fontXHeight/2)
      *bottomDCursor++ = (float)y * xHeightFactor;
     else {
      float temp1 = (float)y * ascenderFactor;
      float temp2 = (float)fontXHeight/2 * xHeightFactor;
      if (temp1<temp2)
          *bottomDCursor++ = temp2;
      else
          *bottomDCursor++ = temp1;
  /*    *bottomDCursor++ = (float)y * ascenderFactor; */
     }
    }

  /* Now try to remove parts of the contour on to the left and right of the
   * word shape that are at height 0 */

  /* Find left edge */
   topDCursor = temp->top;
   bottomDCursor = temp->bottom;
   for (i=0;i<numberOfLegs;++i) {
    if ((*topDCursor++ != 0)||(*bottomDCursor++!=0))
     break;
   }
   left = i;

  /* Find right edge */
   topDCursor = temp->top+numberOfLegs-1;
   bottomDCursor = temp->bottom+numberOfLegs-1;
   for (i=numberOfLegs-1;i>=0;--i) {
    if ((*topDCursor-- != 0)||(*bottomDCursor-- !=0))
     break;
   }
   right = i+1;

  /* Clip the ends of the contour at left and right */
   xDCursor = temp->x;
   topDCursor = temp->top;
   bottomDCursor = temp->bottom;
   xCursor = temp->x+left;
   topCursor = temp->top+left;
   bottomCursor = temp->bottom+left;
   foffset = *xSCursor;
   for (i=left;i<right;++i) {
    *xDCursor++ = *xCursor++ - foffset;
```

```
   *topDCursor++ = *topCursor++;
   *bottomDCursor++ = *bottomCursor++;
  }
  temp->numberOfLegs = right-left;

  *(dict->outlines+dictEntry) = temp;
  ResampleOutlinePair(*(dict->outlines+dictEntry),widthFactor);
}

static int lineSpacing;
int OrderOutlinePair(OutlinePair *o1,OutlinePair *o2)
{
 int yDistance;
 int xDistance;
 yDistance = (*o1)->box->pageY - (*o2)->box->pageY;
 if (yDistance<lineSpacing && yDistance > -lineSpacing) {
  xDistance = (*o1)->box->pageX - (*o2)->box->pageX;
  return xDistance;
 }
 return yDistance;
}

void SortDictionary(Dictionary dict)
{
 lineSpacing = 20;
 qsort(dict->rawOutlines,dict->numberOfEntries,sizeof(RawOutlinePair),
       OrderOutlinePair);
}

#define HIST_SIZE 100
void HistogramMax(int *data,int dataLength,int offset,int sign,int *histogram)
{
 int i,bin;

 if (sign>0) {
  int maxValue;

  maxValue = *data;
  for (i=0;i<dataLength;++i)
   if (data[i]!=HIT_THE_BOX) {
       maxValue = data[i];
       break;
   }
  for (;i<dataLength;++i)
   if (data[i]!=HIT_THE_BOX && data[i]>maxValue)
       maxValue = data[i];
  if (maxValue != HIT_THE_BOX) {
   bin = maxValue-offset;
   if ((bin>=0)&&(bin<HIST_SIZE))
       histogram[bin]++;
  }
 }
 else {
  int minValue;
  minValue = *data;
```

```
   for (i=0;i<dataLength;++i)
    if (data[i]!=HIT_THE_BOX) {
        minValue = data[i];
        break;
    }
   for (;i<dataLength;++i)
    if (data[i]!=HIT_THE_BOX && data[i]<minValue)
        minValue = data[i];
   if (minValue != HIT_THE_BOX) {
    bin = minValue-offset;
    if ((bin>=0)&&(bin<HIST_SIZE))
        histogram[bin]++;
   }
  }
 }

void Histogram(int *data,int dataLength, int offset, int *histogram)
{
 int i,bin;

 for (i=0;i<dataLength;++i) {
  if (*data != HIT_THE_BOX) {
   bin = *data-offset;
   if ((bin>=0)&&(bin<HIST_SIZE))
       histogram[bin]++;
  }
  data++;
 }
}

int MaxBin(int *histogram)
{
 int i;
 int maxValue;
 int maxIndex;

 maxValue = histogram[0];
 maxIndex = 0;
 for (i=0;i<HIST_SIZE;++i)
  if (histogram[i]>maxValue) {
   maxValue = histogram[i];
   maxIndex = i;
  }
 return maxIndex;
}

int MaxBinAbove(int *histogram,int line)
{
 int i;
 int maxValue;
 int maxIndex;
 int top,bottom;

 for (i=0;i<HIST_SIZE;++i)
  if (histogram[i] != 0)
```

```
      break;

    top = i;
    bottom = (line+top)/2;

    maxValue = histogram[top];
    maxIndex = top;
    for (i=top;i<=bottom; ++i)
     if (histogram[i]>maxValue) {
      maxValue = histogram[i];
      maxIndex = i;
     }
    return maxIndex;
   }

   void DrawTextLines(Picture thePict,Dictionary dict,int topLine,int bottomLine)
   {
    int maxLength;
    int halfWidth;
    int x,y;
    float x2,x3,y2,y3;
    float angle;

    angle = (*(dict->rawOutlines))->box->angle;
    maxLength = thePict->width+thePict->height;
    halfWidth = thePict->width / 2;
    x = topLine * -sin(angle) + halfWidth * cos(angle);
    y = topLine * cos(angle) + halfWidth * sin(angle);
    x2 = x+maxLength*cos(angle);
    y2 = y+maxLength*sin(angle);
    x3 = x-maxLength*cos(angle);
    y3 = y-maxLength*sin(angle);
    DrawLine(thePict,x,y,(int)x2,(int)y2,5);
    DrawLine(thePict,x,y,(int)x3,(int)y3,5);

    x = bottomLine * -sin(angle) + halfWidth * cos(angle);
    y = bottomLine * cos(angle) + halfWidth * sin(angle);
    x2 = x+maxLength*cos(angle);
    y2 = y+maxLength*sin(angle);
    x3 = x-maxLength*cos(angle);
    y3 = y-maxLength*sin(angle);
    DrawLine(thePict,x,y,(int)x2,(int)y2,5);
    DrawLine(thePict,x,y,(int)x3,(int)y3,5);
   }

   void PageStatistics(Dictionary dict,char *fileName,NormalizationDescriptor *nd)
   /* WARNING - this must be run before PostProcess since PostProcess changes the raw
    * shape data. */
   {
    int index;
    int temp;
    int i,startIndex,firstY,minY,endIndex,shape;
    int tops[HIST_SIZE];
    int bottoms[HIST_SIZE];
    int ascenders[HIST_SIZE];
```

```
    int descenders[HIST_SIZE];
    int middleLine,topLine,bottomLine,ascenderLine,descenderLine;
    int ascenderHeight,descenderHeight,lineNumber;
    int fontXHeight,fontXWidth,xIndex;
    RawOutlinePair thisShape;
    FILE *fp;
    BOOLEAN haveFirstFontXWidth = FALSE;
    int firstFontXWidth;

    if ((fp=fopen(fileName,"w"))==NULL)
     DoError("PageStatistics: error opening output file %s.\n",fileName);

    SortDictionary(dict);

    index = 0;
#ifdef foo
    malloc_verify();
#endif
    lineNumber = 0;
    while (index < dict->numberOfEntries) {
     startIndex = index;
     firstY = (*(dict->rawOutlines+index))->box->pageY;
     minY = firstY;
     while ( (*(dict->rawOutlines+index))->box->pageY - firstY < 20 &&
             (*(dict->rawOutlines+index))->box->pageY - firstY > -20) {
      if ( minY > ( (*(dict->rawOutlines+index))->box->pageY ))
          minY = (*(dict->rawOutlines+index))->box->pageY;
      ++index;
      if (index == dict->numberOfEntries)
          break;
     }
     endIndex = index;

#ifdef foo
     malloc_verify();
#endif

     /* shapes from start index through endindex are all on */
     /* the same text line */
     /* minY has the top of the highest box on the line. */

     /* Find the base and toplines by taking the mode of the heights of the
      * valleys of the bottom contours and the peaks of the top contours */
     for (i=0;i<HIST_SIZE;i++) {
      tops[i]=0;
      bottoms[i]=0;
      ascenders[i]=0;
      descenders[i]=0;
     }

     for (shape=startIndex;shape<endIndex; ++shape) {
      thisShape = *(dict->rawOutlines+shape);
      Histogram(thisShape->top,thisShape->numberOfLegs,minY,tops);
      Histogram(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);

```

```
      HistogramMax(thisShape->top,thisShape->numberOfLegs,minY,-1,ascenders);
      HistogramMax(thisShape->bottom,thisShape->numberOfLegs,minY,1,descenders);
    }
    topLine = MaxBin(tops)+minY;
    bottomLine = MaxBin(bottoms)+minY;
    ascenderLine = MaxBin(ascenders)+minY;
    descenderLine = MaxBin(descenders)+minY;

    if (thePict)
      DrawTextLines(thePict,dict,topLine,bottomLine);
#ifdef foo
    malloc_verify();
#endif

    middleLine = (bottomLine+topLine)/2;
    fontXHeight = bottomLine-topLine;
    ascenderHeight = bottomLine-ascenderLine;
    if ((float)ascenderHeight/(float)fontXHeight < 1.1) {
      fprintf(stderr,"Bad ascender height on line %d.\n",lineNumber);
      ascenderLine = MaxBinAbove(ascenders,ascenderLine-minY)+minY;
      ascenderHeight = bottomLine-ascenderLine;
      fprintf(stderr,"New ascender height = %d.\nNew xheight =
%d.\n",ascenderHeight,fontXHeight);
    }

    fprintf(fp,"%d: %d %d %2.6f\n",lineNumber,fontXHeight,ascenderHeight,
            (float)ascenderHeight/(float)fontXHeight);

#ifdef foo
    /* Assume that the first shape in the image is the letter x.
     * Use this shape to compute the fontXWidth value. */
    if (lineNumber==0)
      fontXWidth = RawOutlineWidth(*(dict->rawOutlines),middleLine);
#endif

    ++lineNumber;
    if (fontXHeight < 0) {
      fprintf(stderr,"PageStatistics: negative fontXHeight in line %d.\n",lineNumber);
      fontXHeight *= -1;
    }
    for (shape=startIndex;shape<endIndex;++shape)
      StoreOutlinePair(dict,shape,middleLine,fontXHeight,ascenderHeight,nd);
  } /* Do another line of text */
  fclose(fp);
}
```

Jan 12 17:35 1991 getAll.c

```
#include <stdio.h>
#include <math.h>
#include <values.h>
#include "boolean.h"
#include "types.h"
#include "pict.h"
#include "dict.h"

#define MAX_STRING_LEN 256

void WriteShiftedAsciiOutline(FILE *fp, OutlinePair outline, float x, float y)
{
 int i;
 for (i=0;i<outline->numberOfLegs; ++i)
  fprintf(fp,"%f %f\n",i+x,*(outline->top+i)+y);
 fprintf(fp,"\"top\n\n");

 for (i=0;i<outline->numberOfLegs; ++i)
  fprintf(fp,"%f %f\n",i+x,y-(*(outline->bottom+i)));
 fprintf(fp,"\"bottom\n\n");
}

void WriteOutlines(char *filename,Dictionary dict)
{
 float maxWidth,maxHeight;
 int i,j,count;
 int width,height;
 float x,y;
 OutlinePair outline;
 FILE *fp;
 if ((fp = fopen(filename,"w"))==NULL) {
  printf("Error opening %s.",filename);
  exit(-1);
 }

 maxWidth = 0;
 maxHeight = 0;
 for (i=0;i<dict->numberOfEntries; ++i) {
  outline = *(dict->outlines+i);
  if (outline->numberOfLegs > maxWidth)
   maxWidth = outline->numberOfLegs;
  for (j=0;j<outline->numberOfLegs; ++j) {
   if (*(outline->bottom+j)>maxHeight)
       maxHeight = *(outline->bottom+j)>maxHeight;
   if (*(outline->top+j)>maxHeight)
       maxHeight = *(outline->bottom+j)>maxHeight;
  }
 };

 printf("maxWidth,maxHeight = %f,%f\n",maxWidth,maxHeight);

 width = irint(sqrt((double)(dict->numberOfEntries)));
```

```
  height = irint((double)(dict->numberOfEntries) / width);

  printf("n,width,height = %d,%d,%d\n",dict->numberOfEntries,width,height);

  for (i=0;i<height; ++i)
   for (j=0;j<width; ++j) {
    count = i*width+j;
    if ((count < 16) && (count < dict->numberOfEntries)) {
        x = j*maxWidth*1.5;
     y = (height-i+1)*maxHeight*3;
     printf("(%f,%f) ",x,y);
        WriteShiftedAsciiOutline(fp,*(dict->outlines+count),x,y);
    }
   }
  fclose(fp);
}


void main(int argc,char **argv)
{
 char *infile,*outfile;
 Dictionary dict;

 if (argc != 3) {
  printf("Usage:\n");
  printf("  %s infile outfile\n",argv[0]);
  exit(-1);
 }

 infile = argv[1];
 outfile = argv[2];
 dict = ReadDictionary(infile);

 WriteOutlines(outfile,dict);

 printf("\n");
}
```

Jul 8 14:25 1991 getOutline.c

```
#include <stdio.h>
#include <math.h>
#include <values.h>
#include <strings.h>
#include "boolean.h"
#include "types.h"
#include "pict.h"
#include "dict.h"

extern char *strchr(char *s,int c);

#define MAX_STRING_LEN 256

void WriteAsciiOutline(char *filename, OutlinePair outline)
{
 FILE *fp;
 int i;
 if ((fp = fopen(filename,"w"))==NULL) {
  printf("Error opening %s.",filename);
  exit(-1);
 }
 for (i=0;i<outline->numberOfLegs; ++i)
  fprintf(fp,"%d %f\n",i,*(outline->top+i));
 fprintf(fp,"\"top\n\n");

 for (i=0;i<outline->numberOfLegs; ++i)
  fprintf(fp,"%d %f\n",i,-(*(outline->bottom+i)));
 fprintf(fp,"\"bottom\n\n");
 fclose(fp);
}


void main(int argc,char **argv)
{
 char *infile;
 char s[MAX_STRING_LEN],outfile[MAX_STRING_LEN];
 Dictionary dict;
 int selection;
 char *crPointer;
 BOOLEAN done = FALSE;

 if (argc != 2) {
  printf("Usage:\n");
  printf("  %s infile\n",argv[0]);
  exit(-1);
 }

 infile = argv[1];
 dict = ReadDictionary(infile);

 while (!done) {
  printf("Shape number [0..%d]: ",dict->numberOfEntries-1);
```

```
    fgets(s,MAX_STRING_LEN,stdin);
    if (sscanf(s,"%d",&selection) == 1) {
     if (selection<0 || selection>=dict->numberOfEntries)
         printf("Shape numbers must be between 0 and %d, inclusive.\n",
          dict->numberOfEntries-1);
     else {
         printf("Output file: ");
         fgets(outfile,MAX_STRING_LEN,stdin);
         crPointer = strchr(outfile,'\n');
         if (crPointer != NULL)
          *crPointer = '\0';
         printf("Writing shape %d to file %s\n",selection,outfile);
         WriteAsciiOutline(outfile,*(dict->outlines+selection));
     }
    }
    else if ((s[0] == '\0') || (s[0] == '\n'))
     done = TRUE;
    else {
     printf("Enter an integer to select a shape or a blank line\n");
     printf("to quit.\n");
    }
   }
  }
```

Jan 11 17:06 1991 guassian.c

```
#include <stdio.h>
#include <math.h>
#include <values.h>

float square(float x)
{
 return x*x;
}

float gaussian(a, s, x)  /* return A*GAUSS(SIGMA, X) */
float a, s, x;
{
 return (a*exp(-square(x/s)/2.0))/(s*sqrt(2.0*M_PI));
}

float *MakeMask(int halfMaskSize, float a)
{
 int mask_size;
 int x;
 float s;
 float *mask, sum;

 mask_size = 2*halfMaskSize+1;
 s = halfMaskSize/2;
 mask = (float *) calloc(halfMaskSize+1, sizeof(float));
 if (mask == NULL) {
  printf("MakeMask: cannot allocate space\n");
  exit(-1);
 }

 for (x = 0; x <= halfMaskSize; x++) {
  mask[x] = gaussian(a, s, (float) x);
/*   printf("%e\n",mask[x]); */
 }

 for (sum = fabs(mask[0]), x = 1; x <= halfMaskSize; x++)
  sum += 2.0*fabs(mask[x]);

 for (x = 0; x <= halfMaskSize; x++)
  mask[x] /= sum;

 return mask;
}

void Guass1DFloat(float *data, int n, int halfMaskSize)
{
 float a;
 float *mask;
 float *newData;
 float *leftPtr,*rightPtr;
 float sum;
 int i,j,left,right;
```

```
 a=1;

 if (n < halfMaskSize*2+1)
  return;

 newData = (float *)calloc(n,sizeof(float));
 if (newData == NULL) {
  printf("Guass1DFloat: cannot allocate space\n");
  exit(-1);
 }

 mask = MakeMask(halfMaskSize,a);

 for (i=halfMaskSize;i<n-halfMaskSize;++i) {
  sum = *(data+i) * mask[0];
  leftPtr = rightPtr = data+i;
  for (j=1;j<halfMaskSize;++j)
   sum += mask[j] * (*(--leftPtr) + *(++rightPtr));
  newData[i] = sum;
 }

 for (i=0;i<halfMaskSize;++i) {
  sum = data[i]*mask[0];
  left = i;
  right = i;
  for (j=1;j<halfMaskSize;++j) {
   if (--left < 0)
       left += n;
   if (++right >=n)
       right -= n;
   sum += mask[j] * ( data[left] + data[right] );
  }
  newData[i] = sum;
 }

 for (i=n-halfMaskSize;i<n;++i) {
  sum = data[i]*mask[0];
  left = i;
  right = i;
  for (j=1;j<halfMaskSize;++j) {
   if (--left < 0)
       left += n;
   if (++right >=n)
       right -= n;
   sum += mask[j] * ( data[left] + data[right] );
  }
  newData[i] = sum;
 }

 leftPtr = data;
 rightPtr = newData;
 for (i=0;i<n;++i)
  *leftPtr++ = *rightPtr++;
 free(newData);
```

Section D APPENDIX / Page 200

108 }

Aug 23 19:21 1991 lines.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"
#include "lines.h"

void LineEngine(Picture pict,
              int x1,
              int y1,
              int x2,
              int y2,
              UCHAR color,
              pistFunc PerPixel)
{
 static int inside = 0;
 int xinc,yinc;
 int distance;
 int left,right,top,bottom;

 ++inside;
 left = 0;
 right = pict->width-1;
 top = 0;
 bottom = pict->height-1;
 /* printf("Draw line: (%d,%d)-(%d,%d)\n",x1,y1,x2,y2); */
 /* CASE VERTICAL */
 yinc = y2 - y1;
 xinc = x2 - x1;
 if (xinc > 0) {
  if (yinc > 0) {
   /* Line goes up to the right */
   if (yinc>xinc)
       distance = -yinc;
   else
       distance = xinc;
   while ((*PerPixel)(pict,x1,y1,
                  ((x1 < x2) || (y1 < y2))&&(x1<=right)&&(y1<=bottom),
                  color)) {
       if (distance > 0) {
        /* move right */
        x1++;
        distance -= yinc;
       } else {
        /* move up */
        y1++;
        distance += xinc;
       }
   }
  } else {
   if (-yinc>xinc)
       distance = yinc;
```

```
      else
          distance = xinc;
      while ((*PerPixel)(pict,x1,y1,
                      ((x1 < x2) || (y1 > y2))&&(x1<=right)&&(y1>=top),
                      color)) {
          if (distance > 0) {
           /* move right */
           x1++;
           distance += yinc;
          } else {
           /* move down */
           y1--;
           distance += xinc;
          }
      }
    }
  } else {
    if (yinc > 0) {
     /* Line goes up to the left */
     if (yinc>-xinc)
          distance = -yinc;
     else
          distance = -xinc;
     while ((*PerPixel)(pict,x1,y1,
                      ((x1 > x2) || (y1 < y2))&&(x1>=left)&&(y1<=bottom),
                      color)) {
          if (distance > 0) {
           /* move left */
           x1--;
           distance -= yinc;
          } else {
           /* move up */
           y1++;
           distance -= xinc;
          }
     }
    } else {
     if (-yinc>-xinc)
          distance = yinc;
     else
          distance = -xinc;
     while ((*PerPixel)(pict,x1,y1,
                      ((x1 > x2) || (y1 > y2))&&(x1>=left)&&(y1>=top),
                      color)) {
          if (distance > 0) {
           /* move left */
           x1--;
           distance += yinc;
          } else {
           /* move down */
           y1--;
           distance -= xinc;
          }
     }
  }
```

```
  }
  --inside;
}

BOOLEAN DrawPiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
 if (test)
  WriteClippedPixel(pict,x,y,color);
 return test;
}

static UCHAR bitmasks[] = { 0x80,0x40,0x20,0x10,0x8,0x4,0x2,0x1 };

void CountLine1Bit(Picture pict,
                int x1,
                int y1,
                int x2,
                int y2,
                int *totalSet,
                int *total)
{
 static int inside = 0;
 int xinc,yinc;
 int distance;
 int left,right,top,bottom;

 int uchar_width;
 UCHAR *cursor;
 UCHAR mask;
 int count = 0;
 int pixels = 0;

 ++inside;
 left = 0;
 right = pict->width-1;
 top = 0;
 bottom = pict->height-1;

 if (pict->depth != 1)
  DoError("CountLine1Bit: Only depth 1 is supported.\n",NULL);

 uchar_width = pict->uchar_width;
 cursor = pict->data+y1*uchar_width+(x1>>3);
 mask = bitmasks[x1%8];

 /* printf("Draw line: (%d,%d)-(%d,%d)\n",x1,y1,x2,y2); */
 /* CASE VERTICAL */
 yinc = y2 - y1;
 xinc = x2 - x1;
 if (xinc > 0) {
  if (yinc > 0) {
   /* Line goes up to the right */
   if (yinc>xinc)
       distance = -yinc;
   else
```

```
            distance = xinc;
        while (((x1 < x2) || (y1 < y2))&&(x1<=right)&&(y1<=bottom)) {
            if (*cursor & mask)
              ++count;
            ++pixels;
            if (distance > 0) {
              /* move right */
              if (mask == 0x1) {
                mask = 0x80;
                ++cursor;
              }
              else
                mask = mask >> 1;
              x1++;
              distance -= yinc;
            } else {
              /* move up */
              cursor += uchar_width;
              y1++;
              distance += xinc;
            }
        }
      } else {
        if (-yinc>xinc)
            distance = yinc;
        else
            distance = xinc;
        while (((x1 < x2) || (y1 > y2))&&(x1<=right)&&(y1>=top)) {
            if (*cursor & mask)
              ++count;
            ++pixels;
            if (distance > 0) {
              /* move right */
              if (mask == 0x1) {
                mask = 0x80;
                ++cursor;
              }
              else
                mask = mask >> 1;
              x1++;
              distance += yinc;
            } else {
              /* move down */
          cursor -= uchar_width;
              y1--;
              distance += xinc;
            }
        }
      }
    } else {
      if (yinc > 0) {
        /* Line goes up to the left */
        if (yinc>-xinc)
            distance = -yinc;
        else
```

```
          distance = -xinc;
       while (((x1 > x2) || (y1 < y2))&&(x1>=left)&&(y1<=bottom)) {
          if (*cursor & mask)
            ++count;
          ++pixels;
          if (distance > 0) {
           /* move left */
           if (mask == 0x80) {
            mask = 0x1;
            --cursor;
           }
           else
            mask = mask << 1;
           x1--;
           distance -= yinc;
          } else {
           /* move up */
           cursor += uchar_width;
           y1++;
           distance -= xinc;
          }
        }
      } else {
        if (-yinc>-xinc)
          distance = yinc;
        else
          distance = -xinc;
        while (((x1 > x2) || (y1 > y2))&&(x1>=left)&&(y1>=top)) {
          if (*cursor & mask)
            ++count;
          ++pixels;
          if (distance > 0) {
           /* move left */
           if (mask == 0x80) {
            mask = 0x1;
            --cursor;
           }
           else
            mask = mask << 1;
           x1--;
           distance += yinc;
          } else {
           /* move down */
           cursor -= uchar_width;
           y1--;
           distance -= xinc;
          }
        }
      }
    }
    --inside;
    *totalSet += count;
    *total += pixels;
  }
```

```
void DrawLine(Picture pict, int x1, int y1, int x2, int y2, UCHAR color)
{
 LineEngine(pict,x1,y1,x2,y2,color,DrawPiston);
}

static int pixelCounter;
static int setCounter;
BOOLEAN CountPiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
 if (test) {
  ++pixelCounter;
  if (ReadPixel(pict,x,y))
   ++setCounter;
 }
 return test;
}

#ifdef foo
float CountLine(Picture pict, int x1, int y1, int x2, int y2)
{
 pixelCounter = 0;
 setCounter = 0;
 LineEngine(pict,x1,y1,x2,y2,0,CountPiston);
 LineEngine(pict,x1,y1,x1-(x2-x1),y1-(y2-y1),0,CountPiston);
 return (float)setCounter/pixelCounter;
}
#endif

float CountLine(Picture pict, int x1, int y1, int x2, int y2)
{
 pixelCounter = 0;
 setCounter = 0;
 CountLine1Bit(pict,x1,y1,x2,y2,&setCounter,&pixelCounter);
 CountLine1Bit(pict,x1,y1,x1-(x2-x1),y1-(y2-y1),&setCounter,&pixelCounter);
 return (float)setCounter/pixelCounter;
}

static int startx;
static int starty;
static int endx;
static int endy;
BOOLEAN DistancePiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
 if (test) {
  if (ReadPixel(pict,x,y)) {
   if ((x==startx)&&(y==starty))
      return test;
   else {
      endx = x;
      endy = y;
      return FALSE;
   }
  }
  else
   return test;
```

```
  } else
    return test;
}


int DistanceLine(Picture pict, int x1, int y1, int x2, int y2)
{
  double dx,dy;
  startx = x1;
  starty = y1;
  endx = x2;
  endy = y2;
  LineEngine(pict,x1,y1,x2,y2,0,DistancePiston);
  dx = endx-x1;
  dy = endy-y1;
  return sqrt(dx*dx+dy*dy);
}


#ifdef TEST
void draw(pict)
Picture pict;
{
  float angle;
  float step;
  float x1,y1,x2,y2;
  float r1,r2;
  int xc,yc;

  xc = 320;
  yc = 250;
  r1 = 50;
  r2 = 400;
  step = M_PI*2/50;

  for (angle = 0;angle < 2*M_PI; angle += step) {
    x1 = xc + r1*cos(angle);
    y1 = yc + r1*sin(angle);
    x2 = xc + r2*cos(angle);
    y2 = yc + r2*sin(angle);
    DrawLine(pict,(int)x1,(int)y1,(int)x2,(int)y2,0xff);
    printf("%3.2f: %d %d\n",angle,
           CountLine(pict,(int)x1,(int)y1,(int)x2,(int)y2),
           DistanceLine(pict,(int)x1,(int)y1,(int)x2,(int)y2));
  }
}

void main(argc,argv)
int argc;
char **argv;
{
  char *outfile;
  Picture pict;

  if (argc != 2) {
```

```
    printf("Usage: %s outfile\n",argv[0]);
    exit(0);
   }
   outfile = argv[1];

   pict = new_pict(640,500,1);

   draw(pict);

   write_pict(outfile,pict);
   printf("done\n");
  }
  #endif
```

Aug 23 16:43 1991 maxFilter.c

```
#include <stdio.h>
#include "mylib.h"

extern int irint(double);

#define MAX_SIGNAL_LENGTH (10000)
#define MIN_MODE (5) /* MIN_MODE must be less than MAX_HIST_SIZE */
#define MAX_HIST_SIZE (500)
#define MAX_PEAKS (100)
#define BASE_PERCENTILE (0.5)
float data[MAX_SIGNAL_LENGTH];
int newSignal[MAX_SIGNAL_LENGTH];

int MaxOnInterval(int start,int end)
{
 int i;
 float maxValue = data[start];
 int maxIndex = start;
 for (i=start;i<end;++i)
  if (data[i]>maxValue) {
   maxValue = data[i];
   maxIndex = i;
  }
 return maxIndex;
}

void main(int argc,char **argv)
{
 char *infile,*outfile;
 FILE *inFP,*outFP;
 int signalLength;
 float *cursor;
 int foo;
 int i;
 int maskWidth = 10;
 float maxValue;
 int maxIndex,modeValue,modeIndex;
 int h[MAX_HIST_SIZE];
 int finalCount;
 int finalIndex[MAX_PEAKS];
 float baseThresh;
 BOOLEAN upState;
 float thisRatio,lastRatio;

 DefArg("%s %s","infile outfile",&infile,&outfile);
 ScanArgs(argc,argv);

 if ((inFP=fopen(infile,"r"))==NULL)
  DoError("Error opening file %s.\n",infile);

 cursor = data;
 while (fscanf(inFP,"%d %f\n",&foo,cursor++)==2)
```

```
   if (cursor-data>MAX_SIGNAL_LENGTH)
    DoError("Signal is too long.\n",NULL);
  signalLength = cursor-data;

 /* Compute the threhold for the black edge to black pixel ratio */
  maxValue = data[0];
  for (i=0;i<signalLength; ++i) {
   if (data[i]>maxValue)
    maxValue = data[i];
  }
  baseThresh = maxValue*BASE_PERCENTILE;
  printf("baseThresh = %3.3f\n",baseThresh);

 /* Get the indices of the peaks taller than baseThresh */
  finalCount = 0;
  upState = TRUE;
  for (i=0;i<signalLength; ++i) {
   thisRatio = data[i];
   if (thisRatio < baseThresh)
    thisRatio = 0;
   if (upState) {
    if (thisRatio < lastRatio) {
        finalIndex[finalCount] = i;
        finalCount++;
        upState = FALSE;
    }
   }
   else {
    /* upState == FALSE */
    if (thisRatio > lastRatio)
        upState = TRUE;
   }
   lastRatio = thisRatio;
   if (finalCount==MAX_PEAKS)
    break;
  }

 /* Histogram the distances between adjacent peaks */
  for (i=0;i<MAX_HIST_SIZE;h[i++]=0);
  for (i=0;i<finalCount-1; ++i) {
   int d;
   d = finalIndex[i+1]-finalIndex[i];
   if (d<MAX_HIST_SIZE)
    h[d]++;
  }

 /* Find the mode of the adjacent distances that is above MIN_MODE */
  modeValue = h[MIN_MODE];
  modeIndex = MIN_MODE;
  for (i=MIN_MODE;i<MAX_HIST_SIZE; ++i)
   if (h[i]>modeValue) {
    modeValue = h[i];
    modeIndex = i;
   }
```

```
/* Set the mask width to half of the most common spacing of largest peaks */
 maskWidth = irint(modeIndex*0.80);
 printf("maskWidth = %d.\n",maskWidth);

 for (i=0;i<signalLength;newSignal[i++]=0);
 for (i=0;i<signalLength-maskWidth;++i)
  newSignal[MaxOnInterval(i,i+maskWidth)]++;

 if ((outFP=fopen(outfile,"w"))==NULL)
  DoError("Error opening file %s.\n",NULL);
 for (i=0;i<signalLength;++i)
  fprintf(outFP,"%d %d\n",i,newSignal[i]);
 fclose(outFP);
}
```

Jun 19 21:22 1991 myWc.c

```
#include <stdio.h>
#include "boolean.h"
#include "error.h"

typedef int State;
#define WHITE_SPACE 0
#define UNKNOWN_WORD 1
#define ASCENDER_WORD 2


#define MAX_STRING_LENGTH 200

BOOLEAN isWhite(char c)
{
 return (c==' '||c=='\t'||c=='\0'||c=='\n');
}

BOOLEAN isAscender(char c)
{
 return ((c=='b')||(c=='d')||(c=='f')||(c=='h')||(c=='i')||(c=='j')||(c=='k')||(c=='l')||
        (c=='t')||((c>='A')&&(c<='Z'))||((c>='0')&&(c<='9'))||(c=='\'')||(c=='"'));
}

void main(int argc,char **argv)
{
 char *filename;
 FILE *fp;
 char s[MAX_STRING_LENGTH+1];
 char *ptr;
 State state;
 int wordsWithAscenders,wordsWithoutAscenders,words;

 if (argc != 2) {
  fprintf(stderr,"Usage:\n");
  fprintf(stderr,"  %s <input file>\n");
  exit(-1);
 }

 filename = argv[1];
 if ((fp=fopen(filename,"r"))==NULL)
  DoError("%s: cannot open input file.\n",filename);

 wordsWithAscenders = 0;
 wordsWithoutAscenders = 0;
 words = 0;
 fgets(s,MAX_STRING_LENGTH,fp);
 while (!feof(fp)) {
  ptr = s;
  state = WHITE_SPACE;
  while (*ptr != '\0') {
   switch (state) {
    case WHITE_SPACE:
```

```
            if (isWhite(*ptr))
             ++ptr;
            else
             state = UNKNOWN_WORD;
            break;
        case UNKNOWN_WORD:
            if (isWhite(*ptr)) {
             ++wordsWithoutAscenders;
             ++words;
             state = WHITE_SPACE;
            }
            if (isAscender(*ptr)) {
             ++wordsWithAscenders;
             ++words;
             ++ptr;
             state = ASCENDER_WORD;
            }
            else
             ++ptr;
            break;
        case ASCENDER_WORD:
            if (isWhite(*ptr))
             state = WHITE_SPACE;
            ++ptr;
            break;
           default:
            DoError("myWc: internal error - bad state.\n",NULL);
      } /* switch */
     } /* while (*ptr ... */
     fgets(s,MAX_STRING_LENGTH,fp);
    } /* while (!eof ... */
    printf("words: %d\n",words);
    printf("words with ascenders: %d\n",wordsWithAscenders);
    printf("words without ascenders: %d\n",wordsWithoutAscenders);
    printf("word ascender/descender ratio: %6.2f\n",
          (float)wordsWithAscenders/(float)wordsWithoutAscenders);
   }
```

Aug 23 18:12 1991 newBaselines.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"
#include "types.h"
#include "lists.h"
#include "lines.h"
#include "baselines.h"

extern double sqrt(double);
extern int irint(double);

/*inline*/ int NewReadPixel(UCHAR *base,int width,float x,float y)
{
 int xi;
 int yi;
 UCHAR mask;

 xi = irint(x);
 yi = irint(y);
 mask = 0x80 >> (xi & 0x7);
 return *(base+yi*width+(xi>>3)) & mask;
}

void NewCountLine1Bit(Picture pict,int x1,int y1,int x2,int y2,int *black,int *blackEdge)
{
 float x,y;
 float xinc,yinc;
 float xupinc,yupinc;
 float den;
 int b,be;
 int width,ucharWidth,height;
 UCHAR *data;

 width = pict->width;
 ucharWidth = pict->uchar_width;
 height = pict->height;
 data = pict->data;

 den = sqrt((y2-y1)*(y2-y1)+(x2-x1)*(x2-x1));
 xinc = (x2-x1)/den;
 yinc = (y2-y1)/den;
 xupinc = -yinc;
 yupinc = xinc;
 x = x1;
 y = y1;

 b=0;
 be=0;

 while (x<width&&x>=0&&y<height&&y>=0) {
```

```
      ++b;
      if (NewReadPixel(data,ucharWidth,x,y)) {
       if (!(NewReadPixel(data,ucharWidth,x+xupinc,y+yupinc) &&
             NewReadPixel(data,ucharWidth,x-xupinc,y-yupinc)))
          ++be;
      }
      x += xinc;
      y += yinc;

    }
    *black = b;
    *blackEdge = be;
   }


   #define MIN_BLACK 5
   void NewCountLine(Picture pict,int x1,int y1,int x2,int y2,int *black,int *blackEdge)
   {
    *black = 0;
    *blackEdge = 0;
    NewCountLine1Bit(pict,x1,y1,x2,y2,black,blackEdge);
    NewCountLine1Bit(pict,x1,y1,x1-(x2-x1),y1-(y2-y1),black,blackEdge);
   }

   static float x2offset;
   static float y2offset;
   static int projectIndex;
   static int *blackPixels;
   static int *blackEdgePixels;
   static int *coordx;
   static int *coordy;
   BOOLEAN BaseLinePiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
   {
    if (test) {
     NewCountLine(pict,x,y,(int)(x+x2offset),(int)(y+y2offset),
                 blackPixels+projectIndex,blackEdgePixels+projectIndex);
     coordx[projectIndex] = x;
     coordy[projectIndex++] = y;
     return test;
    } else
     return test;
   }

   static int lastX;
   static int lastY;
   BOOLEAN EndPointPiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
   {
    if (test) {
     lastX = x;
     lastY = y;
    }
    return test;
   }

   void EndPoints(Picture pict,double angle,int *tx, int *ty,int *bx, int *by)
```

```
{
 int xc,yc;
 int maxLength;
 float normal;
 float x2,y2,x3,y3;

 /* Make normal to text point in quadrants I and II */
 /* Assume 0 <= angle < 2*M_PI */
 normal = fmod(angle + M_PI/2,2*M_PI);
 if (normal > M_PI)
  normal -= M_PI;

 xc = pict->width/2;
 yc = pict->height/2;

 maxLength = pict->width+pict->height;
 x2 = xc+maxLength*cos(normal);     /* At bottom of picture */
 y2 = yc+maxLength*sin(normal);
 x3 = xc-maxLength*cos(normal);     /* At top of picture */
 y3 = yc-maxLength*sin(normal);

 LineEngine(pict,xc,yc,(int)x2,(int)y2,0,EndPointPiston);
 *bx = lastX;
 *by = lastY;
 LineEngine(pict,xc,yc,(int)x3,(int)y3,0,EndPointPiston);
 *tx = lastX;
 *ty = lastY;
}

double distance(int x1,int y1,int x2,int y2)
{
 return sqrt((double)((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)));
}

FILE *PlotBaselineContour(char *plotFile,int topCount,
                          float *ratios,int *newSignal,
                          float baseThresh)
{
 FILE *outfile;
 int i;

 printf("Opening baselines plot file\n");
 if ((outfile = fopen(plotFile,"w"))==NULL) {
  printf("Error opening baseline plot file.\n");
  exit(-1);
 }
 for (i=0;i<topCount;++i)
  fprintf(outfile,"%d %f\n",i,ratios[i]/baseThresh*5);
 fprintf(outfile,"\"Ratio\n\n");
 for (i=0;i<topCount;++i)
  fprintf(outfile,"%d %d\n",i,newSignal[i]);
 fprintf(outfile,"\"Projection\n\n");
 fprintf(outfile,
        "0 %f\n%d %f%\n\"Baseline Threshold\n",
        baseThresh,topCount,baseThresh);
```

```
 return outfile;
}

int MaxOnInterval(float *data,int start,int end)
{
 int i;
 float maxValue = data[start];
 int maxIndex = start;
 for (i=start;i<end;++i)
  if (data[i]>maxValue) {
   maxValue = data[i];
   maxIndex = i;
  }
 return maxIndex;
}

#define BASE_PERCENTILE 0.50
#define MIN_LINE_HEIGHT_FRACTION 0.50
#define MIN_MODE (5) /* MIN_MODE must be less than MAX_HIST_SIZE */
#define MAX_HIST_SIZE (500)
#define MAX_BASELINES (300)
List BaseLines(Picture pict,double angle,char *plotFile)
{
 float *topProjection;
 int *topCoordx,*topCoordy;
 int *finalCoordx,*finalCoordy,*finalIndex;
 int topIndex,bottomIndex;
 int topCount,botCount,finalCount;
 int maxLength;
 int xc,yc;
 float x2,y2,x3,y3;
 float maxValue,lastValue;
 int i,j;
 float baseThresh;
 int topX,topY,bottomX,bottomY;
 BOOLEAN onTextLine;
 List xList,yList,result;
 double totalDistance,averageDistance;
 FILE *outfile;
 int inside;
 BOOLEAN upState;
 float ratio,lastRatio,thisRatio;
 float *ratios;
 int *newSignal;
 int halfMaskWidth = 10; /* for computing ratios */
 int maxIndex,modeValue,modeIndex;
 int h[MAX_HIST_SIZE];
 int maskWidth; /* for max filter */

 printf("angle = %3.3f\n",angle);

/* The longest ling though the picture will be shorter than maxLength */
 maxLength = pict->width+pict->height;

/* Allocate space for the page projection values */
```

```
    blackPixels = (int *)calloc(maxLength,sizeof(int));
    blackEdgePixels = (int *)calloc(maxLength,sizeof(int));
    ratios = (float *)calloc(maxLength,sizeof(float));
    newSignal = (int *)calloc(maxLength,sizeof(int));
    topCoordx = (int *)calloc(maxLength,sizeof(int));
    topCoordy = (int *)calloc(maxLength,sizeof(int));
    finalCoordx = (int *)calloc(maxLength,sizeof(int));
    finalCoordy = (int *)calloc(maxLength,sizeof(int));
    finalIndex = (int *)calloc(maxLength,sizeof(int));

    if ((blackPixels == NULL)||
      (blackEdgePixels == NULL)||
      (ratios == NULL)||
      (newSignal==NULL)||
      (topCoordx == NULL)||
      (topCoordy == NULL)||
      (finalIndex == NULL) ||
      (finalCoordx == NULL) ||
      (finalCoordy == NULL)) {
     printf("BaseLines: cannot allocate memory\n");
     exit(-1);
    }

   /* Compute the endpoints of a line through the center of the picture in the direction
    * perpendicular to the text lines. This line will be used as the reference frame for
    * computing projections. */
    EndPoints(pict,angle,&topX,&topY,&bottomX,&bottomY);

    printf("Main Line: (%d,%d)-(%d,%d)\n",topX,topY,bottomX,bottomY);
   /* DrawLine(pict,topX,topY,bottomX,bottomY,0xff); */

   /* Compute the projection of the image at each point along the line.
    * topProjection will have the number of black pixels on a line and
    * ratios will have the fraction of black pixels on a line that are
    * the ends of vertical extents. */
    x2offset = maxLength*cos(angle);
    y2offset = maxLength*sin(angle);
    projectIndex = 0;
    coordx = topCoordx;
    coordy = topCoordy;
    LineEngine(pict,topX,topY,bottomX,bottomY,0,BaseLinePiston);
    topCount = projectIndex;

   /* Compute the ratios plot */
    for (i=0;i<halfMaskWidth;++i)
     ratios[i] = 0;
    for (i=topCount-halfMaskWidth;i<topCount;++i)
     ratios[i] = 0;
    for (i=0,inside=0;i<halfMaskWidth*2+1;++i)
     inside += blackPixels[i];
    for (i=halfMaskWidth;i<topCount-halfMaskWidth;++i) {
     ratios[i] = (float)blackEdgePixels[i]/inside;
     inside -= blackPixels[i-halfMaskWidth];
     inside += blackPixels[i+halfMaskWidth];
    }
```

```
/* Compute the threhold for the black edge to black pixel ratio */
 maxValue = ratios[0];
 for (i=0;i<topCount;++i) {
  if (ratios[i]>maxValue)
   maxValue = ratios[i];
 }

 baseThresh = maxValue*BASE_PERCENTILE;
 printf("baseThresh = %3.3f\n",baseThresh);

/* Get the coordinates of the baselines and toplines by finding peaks in the
 * ratios projection. */
 finalCount = 0;
 upState = TRUE;
 for (i=0;i<topCount;++i) {
  thisRatio = ratios[i];
  if (thisRatio < baseThresh)
   thisRatio = 0;
  if (upState) {
   if (thisRatio < lastRatio) {
       finalIndex[finalCount] = i;
       finalCount++;
       upState = FALSE;
   }
  }
  else {
   /* upState == FALSE */
   if (thisRatio > lastRatio)
       upState = TRUE;
  }
  lastRatio = thisRatio;
  if (finalCount == MAX_BASELINES) {
   fprintf(stderr,"Warning: found too many baselines.\n");
   fprintf(stderr,"Ignoring remaining baselines.\n");
   break;
  }
 }

/* Histogram the distances between adjacent peaks */
 for (i=0;i<MAX_HIST_SIZE;h[i++]=0);
 for (i=0;i<finalCount-1;++i) {
  int d;
  d = finalIndex[i+1]-finalIndex[i];
  if (d<MAX_HIST_SIZE)
   h[d]++;
 }

/* Find the mode of the adjacent distances that is above MIN_MODE */
 modeValue = h[MIN_MODE];
 modeIndex = MIN_MODE;
 for (i=MIN_MODE;i<MAX_HIST_SIZE;++i)
  if (h[i]>modeValue) {
   modeValue = h[i];
   modeIndex = i;
```

```
   }

 /* Set the mask width to half of the most common spacing of largest peaks */
  maskWidth = irint(modeIndex*0.80);
  printf("maskWidth = %d.\n",maskWidth);

  for (i=0;i<topCount;newSignal[i++]=0);
  for (i=0;i<topCount-maskWidth;++i)
   newSignal[MaxOnInterval(ratios,i,i+maskWidth)]++;

 /* Plot the baseline contour if requested */
  if (plotFile!=NULL)
   outfile = PlotBaselineContour(plotFile,topCount,ratios,newSignal,baseThresh);

 /* Pick off the new peaks */
 /* Compute the threhold for the black edge to black pixel ratio */
  maxValue = newSignal[0];
  for (i=0;i<topCount;++i) {
   if (newSignal[i]>maxValue)
    maxValue = newSignal[i];
  }

  baseThresh = maxValue*0.80;
  printf("baseThresh = %3.3f\n",baseThresh);

 /* Get the coordinates of the baselines and toplines by finding peaks in the
  * ratios projection. */
  finalCount = 0;
  upState = TRUE;
  for (i=0;i<topCount;++i) {
   thisRatio = newSignal[i];
   if (thisRatio < baseThresh)
    thisRatio = 0;
   if (upState) {
    if (thisRatio < lastRatio) {
        finalCoordx[finalCount] = topCoordx[i];
        finalCoordy[finalCount] = topCoordy[i];
        finalIndex[finalCount] = i;
        finalCount++;
        upState = FALSE;
    }
   }
   else {
    /* upState == FALSE */
    if (thisRatio > lastRatio)
        upState = TRUE;
   }
   lastRatio = thisRatio;
   if (finalCount == MAX_BASELINES) {
     fprintf(stderr,"Warning: found too many baselines.\n");
     fprintf(stderr,"Ignoring remaining baselines.\n");
     break;
   }
  }
```

```
/* ------------------------------------------------------------------------ */

  if (finalCount&1)
   --finalCount;          /* Only take an even number of lines */
  for (totalDistance=0,i=0,j=0;i<finalCount;i+=2) {
   topX = finalCoordx[i];
   topY = finalCoordy[i];
   bottomX = finalCoordx[i+1];
   bottomY = finalCoordy[i+1];
   totalDistance += distance(topX,topY,bottomX,bottomY);
   j+=2;
  }
  averageDistance = totalDistance / (finalCount/2)*MIN_LINE_HEIGHT_FRACTION;
  for (i=0,j=0;i<finalCount;i+=2) {
   topX = finalCoordx[i];
   topY = finalCoordy[i];
   topIndex = finalIndex[i];
   bottomX = finalCoordx[i+1];
   bottomY = finalCoordy[i+1];
   bottomIndex = finalIndex[i+1];
   finalCoordx[j] = topX;
   finalCoordy[j] = topY;
   finalIndex[j] = topIndex;
   finalCoordx[j+1] = bottomX;
   finalCoordy[j+1] = bottomY;
   finalIndex[j+1] = bottomIndex;
   if (distance(topX,topY,bottomX,bottomY)>averageDistance)
    j+=2;
  }
#ifdef foo
 *count = j;
 *returnCoordx = finalCoordx;
 *returnCoordy = finalCoordy;
#endif
 result = nil;
 for (i=j-1;i>=0;--i) {
  push(MakePoint(finalCoordx[i],finalCoordy[i]),result);
 }

 if (plotFile != NULL) {
  fprintf(outfile,"\n0 %f\n",-baseThresh);
  for (i=0;i<j;i+=2) {
       fprintf(outfile,"%d %f\n%d %f\n%d %f\n%d %f\n",
              finalIndex[i],-baseThresh,
              finalIndex[i],-2*baseThresh,
              finalIndex[i+1],-2*baseThresh,
              finalIndex[i+1],-baseThresh);
  }
  fprintf(outfile,"\"Baselines");
  fclose(outfile);
  printf("Done writing baseline plot file.\n");
 }

 return result;
```

```
}

void DrawBaseLines(Picture pict, List pointList, double angle)
#ifdef foo
int count,int *coordx,int *coordy,double angle)
#endif
{
 int maxLength;
 float x2,y2,x3,y3;
 int x,y;
 Point temp;
 maxLength = pict->width+pict->height;
 while (!endp(pointList)) {
  temp = pop(pointList);
  x = temp->x;
  y = temp->y;
  x2 = x+maxLength*cos(angle);
  y2 = y+maxLength*sin(angle);
  x3 = x-maxLength*cos(angle);
  y3 = y-maxLength*sin(angle);
  DrawLine(pict,x,y,(int)x2,(int)y2,0xff);
  DrawLine(pict,x,y,(int)x3,(int)y3,0xff);
 }
}
```

Aug 25 19:48 1991 newBlobify.c

```
#include <stdio.h>
#include <math.h>
#include "mylib.h"
#include "blobify.h"


#define MAX_KERNAL_SIZE (40)

extern int irint(double);

static UCHAR bitmasks[] = {0x80,0x40,0x20,0x10,0x8,0x4,0x2,0x1};

UCHAR *address(Picture pict,float x,float y)
{
 return pict->data+irint(y)*pict->uchar_width+(irint(x)>>3);
}

UCHAR mask(float x)
{
 static masks[] = { 0x80,0x40,0x20,0x10,8,4,2,1};
 return masks[irint(x)%8];
}

int X(float x)
{
 return irint(x);
}

int Y(float y)
{
 return irint(y);
}

Picture NewBlobify(Picture old,int halfMaskWidth,double threshold,double angle)
{
 Picture new;

 int index;
 float x,y,xinc,yinc;
 UCHAR *kernalPtr[MAX_KERNAL_SIZE],*kp[MAX_KERNAL_SIZE];
 UCHAR kernalMask[MAX_KERNAL_SIZE],km[MAX_KERNAL_SIZE];
 int kernalX[MAX_KERNAL_SIZE],kernalY[MAX_KERNAL_SIZE];
 int kx[MAX_KERNAL_SIZE],ky[MAX_KERNAL_SIZE];
 UCHAR kb[MAX_KERNAL_SIZE];
 UCHAR *dest;
 UCHAR dm;
 int tval,i,j,k,inside;
 int width,height,ucharWidth,maskWidth;

 if (halfMaskWidth*2+1 > MAX_KERNAL_SIZE)
  DoError("Blobify: mask is too large.\n",NULL);

```

```
tval = irint(threshold*(halfMaskWidth*2+1));

width = old->width;
height = old->height;
ucharWidth = old->uchar_width;

new = new_pict(width,height,1);

xinc = cos(angle);
yinc = sin(angle);
index = 0;
kernalPtr[index] = address(old,halfMaskWidth,halfMaskWidth);
/*
kernalX[index] = X(halfMaskWidth);
kernalY[index] = Y(halfMaskWidth);
*/
kernalMask[index++] = mask(halfMaskWidth);
for (i=0,x=0,y=0;i<halfMaskWidth;++i) {
 x+=xinc;
 y+=yinc;
 kernalPtr[index] = address(old,halfMaskWidth+x,halfMaskWidth+y);
/*
 kernalX[index] = X(halfMaskWidth+x);
 kernalY[index] = Y(halfMaskWidth+y);
*/
 kernalMask[index++] = mask(halfMaskWidth+x);
 kernalPtr[index] = address(old,halfMaskWidth-x,halfMaskWidth-y);
/*
 kernalX[index] = X(halfMaskWidth-x);
 kernalY[index] = Y(halfMaskWidth-y);
*/
 kernalMask[index++] = mask(halfMaskWidth-x);
}

maskWidth = 2*halfMaskWidth+1;

for (j=0;j<height-maskWidth;++j) {
 for (i=0;i<index;++i) {
  kp[i] = kernalPtr[i]+j*ucharWidth;
  km[i] = kernalMask[i];
  kb[i] = *kp[i]++;
/*
  kx[i] = kernalX[i];
  ky[i] = kernalY[i]+j;
*/
 }
 dest = new->data+(j+halfMaskWidth)*ucharWidth+(halfMaskWidth>>3);
 dm = mask(halfMaskWidth);

 for (k=0;k<width-maskWidth;++k) {
  if (dm == 0) {
     dm = 0x80;
     dest++;
  }
  for (i=0,inside=0;i<index;++i) {
```

```
            if (km[i] == 0) {
             km[i] = 0x80;
             kb[i] = *kp[i]++;
            }
/*
            printf("(%d,%d): %d - %x %x -> %x\n",kx[i],ky[i],kb[i]&km[i],kp[i]-1,km[i],kb[i]);
            kx[i]++;
*/
            if (kb[i]&km[i])
             ++inside;
            km[i] >>= 1;
        }
/*
        printf("%d\n\n",inside);
*/
        if (inside > tval)
            *dest |= dm;
        dm >>= 1;
      }

     }

     return new;

}

#ifdef TRYMAIN
void main(argc,argv)
int argc;
char **argv;
{
 char *infile,*outfile;
 Picture old,new;
 int halfMaskSize;
 float threshold;
 float angle;

 DefArg("%s %s %d %f %f","infile outfile halfMaskSize threshold angle",
        &infile,&outfile,&halfMaskSize,&threshold,&angle);
 ScanArgs(argc,argv);

 printf("Loading %s...",infile);
 old = load_pict(infile);
 new = NewBlobify(old,halfMaskSize,threshold,angle);
 write_pict(outfile,new);
}
#endif
```

Aug 15 06:41 1991 newContour.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "pict.h"
#include "lines.h"
#include "lists.h"
#include "dict.h"
#include "diff.h"
#include "fontNorm.h"

extern Picture thePict; /* Picture used for annotated shapes */

/* The following are misc. definitionas and routines havine to do with
 * vectors and coordinates. */

typedef struct {
double x;
double y;
} DPointBody,*DPoint;


static double Dot(DPoint a,DPoint b)
{
/* printf("Dot: (%lf,%lf)*(%lf,%lf) = %lf\n",a->x,a->y,b->x,b->y,a->x*b->x +
a->y*b->y); */
 return a->x*b->x + a->y*b->y;
}

static DPoint PolarToCartesian(double angle,double radius)
{
 DPoint result = (DPoint)calloc(1,sizeof(DPointBody));
 if (result == NULL)
  DoError("Dot: cannot allocate space\n");
 result->x = cos(angle);
 result->y = sin(angle);
 return result;
}

static DPoint Normal(DPoint a)
{
 DPoint result = (DPoint)calloc(1,sizeof(DPointBody));
 if (result == NULL)
  DoError("Dot: cannot allocate space\n");
 result->x = -a->y;
 result->y = a->x;
 return result;
}
```

```
/* This piston scans pict up and down from the top and bottom of the
 * bounding box, looking for the highest and lowest pixels in the
 * word. If thePict is not NULL, these pixels will be colored as 4
 * in thePict. */
static int startX;
static int startY;
static double stopDistance;
static int lastY;
static BOOLEAN valid;
BOOLEAN TracePiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
 double distance;
 if (test) {
  distance = sqrt((double)(startX-x)*(startX-x)+(startY-y)*(startY-y));
  if (distance<stopDistance) {
/*    lastY = stopDistance - distance; */
   lastY = distance;

   if (ReadPixel(pict,x,y)) {

       if (thePict)
        WritePixel(thePict,x,y,4);

       valid = TRUE;
       return FALSE;
   } else {
       valid = FALSE;
       return test;
   }
  }
  else {
   if (thePict)
       WritePixel(thePict,x,y,4);
#ifdef foo
   lastY = distance;  /**** Used to be 0 ****/
#endif
   lastY = HIT_THE_BOX;
   valid = FALSE;
   return FALSE;
  }
 }
 return test;
}


/* This piston moves from left to right across a bounding box, calling
 * trace piston and saving its output in topY, baseY, and bothX. */
#define MAX_SHELL_LENGTH 400
static int numberOfLegs;
static int topY[MAX_SHELL_LENGTH];
static int baseY[MAX_SHELL_LENGTH];
static int bothX[MAX_SHELL_LENGTH];

static double leftDistance;
static DPoint lineVector;
```

```
static int downX;
static int downY;
static double boxTopDistance;
static double boxBaseDistance;
BOOLEAN ShellPiston(Picture pict, int x, int y, BOOLEAN test, UCHAR color)
{
 int xDistance;
 DPointBody thisPoint;
 if (test) {
  if (numberOfLegs >= MAX_SHELL_LENGTH)
   return FALSE;
  thisPoint.x = x;
  thisPoint.y = y;
  xDistance = Dot(&thisPoint,lineVector) - leftDistance;
  stopDistance = boxTopDistance;
  startX = x;
  startY = y;
  LineEngine(pict,x,y,x+downX,y+downY,0,TracePiston);
  bothX[numberOfLegs] = xDistance;
  if (valid)
   topY[numberOfLegs] = lastY;
  else
   topY[numberOfLegs] = HIT_THE_BOX;

  stopDistance = boxBaseDistance;
  startX = x+downX;
  startY = y+downY;
  LineEngine(pict,x+downX,y+downY,x,y,0,TracePiston);
  if (valid)
   baseY[numberOfLegs] = lastY;
  else
   baseY[numberOfLegs] = HIT_THE_BOX;
  numberOfLegs++;

 }
 return test;
}

/* This function, finds the upper and lower contours corresponding
 * to a word within a bounding box. */
void MakeShell(Picture pict,Box box,
          Dictionary dict, int dictEntry)
{
 DPoint normalVector;
 DPointBody temp;
 double boxTop,boxBase;
 int rightX,rightY;

 lineVector = PolarToCartesian(box->angle,1);
 normalVector = Normal(lineVector);
 temp.x = box->x;
 temp.y = box->y;
 boxTop = Dot(&temp,normalVector);
 box->pageY = irint(boxTop);
 boxBase = boxTop + box->height;
```

```
/* CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE */
 boxTopDistance = boxBase - boxTop;
 boxBaseDistance = boxBase - boxTop;
/* CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE */

 downX = box->height*cos(box->angle+M_PI/2);
 downY = box->height*sin(box->angle+M_PI/2);

 rightX = box->width*cos(box->angle);
 rightY = box->width*sin(box->angle);

 numberOfLegs = 0;
 leftDistance = Dot(&temp,lineVector);
 box->pageX = irint(leftDistance);
#ifdef foo
 malloc_verify();
#endif
 LineEngine(pict,box->x,box->y,
         box->x+rightX,box->y+rightY,0,
         ShellPiston);

/* CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE */
 {
  int i;
  for (i=0;i<numberOfLegs;++i) {
   if (*(topY+i)!=HIT_THE_BOX)
       *(topY+i) += boxTop;
   if (*(baseY+i)!=HIT_THE_BOX)
       *(baseY+i) = boxBase - *(baseY+i);
  }
 }
/* CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE CHANGE */

#ifdef foo
 malloc_verify();
#endif
 StoreRawOutlinePair(dict,dictEntry,box,bothX,topY,
                baseY,numberOfLegs);
}

BOOLEAN OnABaseLine(Box box,List baseLinePoints)
{
 DPoint lineVector,normalVector;
 DPointBody temp;
 double boxTop,boxBase,top,base;
 Point topPoint, basePoint;

 lineVector = PolarToCartesian(box->angle,1);
 normalVector = Normal(lineVector);
 temp.x = box->x;
 temp.y = box->y;
 boxTop = Dot(normalVector,&temp);
 boxBase = boxTop+box->height;
```

```
 while (!endp(baseLinePoints)) {
  topPoint = pop(baseLinePoints);
  basePoint = pop(baseLinePoints);
  temp.x = topPoint->x;
  temp.y = topPoint->y;
  top = Dot(normalVector,&temp);
  temp.x = basePoint->x;
  temp.y = basePoint->y;
  base = Dot(normalVector,&temp);

  if ((boxTop>=top && boxTop <= base) ||  /* box top is between */
       (boxBase>=top && boxBase <= base) || /* box bottom is between */
       (top >= boxTop && top <= boxBase))  /* both lines inside box */
   return TRUE;
 }
 return FALSE;
}

BOOLEAN BoxToShell(Picture pict,Box box,List baseLinePoints,
                Dictionary dict,int dictEntry)
{
 Point topPoint,bottomPoint;

 if (OnABaseLine(box,baseLinePoints)) {
  MakeShell(pict,box,dict,dictEntry);
  return TRUE;
 }
 else
  return FALSE;
}

#define MAX_SHAPES 1000
void BarBoxList(Picture pict,List boxList,List baseLinePoints,
              char *filename,char *infoString, NormalizationDescriptor *nd)
{
 Dictionary dict;
 int count = 0;
 long int location;

 dict = NewDict(MAX_SHAPES);
 dict->infoString = infoString;

 while (!endp(boxList)) {
#ifdef foo
  if (BoxToShell(pict,
              (Box)pop(boxList),
              baseLinePoints,
              dict,
              count))
    ++count;
#endif
/* Change 8/8/91
 * All boxes are stored in the dictionary.
 * The post processing stage in newFontNorm.c will weed out boxes */
```

```
    MakeShell(pict,(Box)pop(boxList),dict,count);
    ++count;
  /* End of change 8/8/91 */
    if (count>=MAX_SHAPES) {
     printf("Maximum dictionary size exceeded.\n");
     printf("Ignoring rest of shapes.\n");
     break;
    }
   }
   dict->numberOfEntries = count;
   PageStatistics(dict,"statistics",nd);
  /* PostProcess(dict); */
   WriteDictionary(dict,filename);
  }
```

Jan 11 17:07 1991 newDiff2.c

```
#include <stdio.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"
#include "diff.h"

/* Given the names of two dictionary files, compute the squared difference
 * between every pair of shapes in the cross product of the dictionaries.
 * The result is a matrix printed to stdout. The width and height are
 * followed by the matrix entries in row major order. The output is in
 * ascii to facilitate reading by a Symbolics. */
Picture CompareDictionaries(char *file1,char *file2)
{
 Dictionary dict1,dict2;
 Picture pict;
 int x,y;
 dict1 = ReadDictionary(file1); /* height */
 dict2 = ReadDictionary(file2); /* width */
 pict = new_pict(dict2->numberOfEntries,
              dict1->numberOfEntries,
              32);
 for (y=0;y<pict->height;++y)
  for (x=0;x<pict->width;++x) {
   printf("(%d,%d) ",y,x);
   *((float *)(pict->data)+pict->width*y+x) =
       DiffPair(*(dict1->outlines+y),
              *(dict2->outlines+x));
  }
 return pict;
}

void WritePictureAsAscii(Picture pict,char *filename)
{
 FILE *fp;
 int x,y;
 int count=1;
 if ((fp = fopen(filename,"w"))==NULL)
  DoError("WritePictureAsAscii: error opening output file\n",NULL);
 fprintf(fp,"%d\n%d\n",pict->width,pict->height);
 for (y=0;y<pict->height;++y)
  for (x=0;x<pict->width;++x) {
   fprintf(fp,"%f ",*(((float *)pict->data)++));
   if (!((count++)%5))
       fprintf(fp,"\n");
  }
 fprintf(fp,"\n");
 fclose(fp);
}
```

Aug 26 17:20 1991 newMain.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "misc.h"
#include "boolean.h"
#include "error.h"
#include "types.h"
#include "pict.h"
#include "lists.h"
#include "lines.h"
#include "orient.h"
#include "baselines.h"
#include "blobify.h"
#include "boxes.h"
#include "dict.h"
#include "diff.h"
#include "newContour.h"
#include "numbers.h"

#define TRY
#ifdef TRY
Picture thePict;
#endif

void DrawMiddleLines(Picture pict,List pointList, double angle)
{
 int maxLength;
 int xc,yc,xBot,xTop,yBot,yTop;
 Point temp;
 float x2,y2,x3,y3;
 int i,len;
 maxLength = pict->width+pict->height;
 len = ListLength(pointList);
 pop(pointList);
 for (i=1;i<len-1;i+=2) {
  temp = pop(pointList);
  xTop = temp->x;
  yTop = temp->y;
  temp = pop(pointList);
  xBot = temp->x;
  yBot = temp->y;
  xc = (xBot+xTop)/2;
  yc = (yBot+yTop)/2;
  x2 = xc+maxLength*cos(angle);
  y2 = yc+maxLength*sin(angle);
  x3 = xc-maxLength*cos(angle);
  y3 = yc-maxLength*sin(angle);
  DrawLine(pict,xc,yc,(int)x2,(int)y2,0);
  DrawLine(pict,xc,yc,(int)x3,(int)y3,0);
 }
}
```

```
void DrawBoxList(Picture pict,List boxList)
{
 while (!endp(boxList)) {
  DrawBox(pict,(Box)pop(boxList));
 }
}

void LabelShapes(Picture pict,Dictionary dict)
{
 int i;
 Box box;

 for (i=0;i<dict->numberOfEntries;++i) {
  box = (*(dict->outlines+i))->box;
  DrawColorBox(pict,box,3);
  DrawNumber(pict,box->x,box->y,2,(float)box->height/2,i);
 }
}

double FixAngle(double angle)
{
 if (angle > M_PI/2 && angle < 1.5*M_PI)
  return angle-M_PI;
 else
  return angle;
}

int ScanIntArg(int argc,char **argv,int index)
{
 if (index<argc)
  return atoi(argv[index]);
 else
  DoError("Expected an integer argument\n",NULL);
}

float ScanFloatArg(int argc,char **argv,int index)
{
 if (index<argc)
  return atof(argv[index]);
 else
  DoError("Expected a floating point argument\n",NULL);
}

char *ScanStringArg(int argc,char **argv,int index)
{
 if (index<argc)
  return argv[index];
 else
  DoError("Expected a string argument\n",NULL);
}

void main(argc,argv)
int argc;
char **argv;
{
```

```
char *infile;
int coarseDirections,coarseSamples,fineDirections,fineSamples;
Picture pict,newPict,finalPict;
float coarseAngle,mediumAngle,fineAngle;
float coarseError,mediumError,fineError;
List baselines,boxList;
int maskWidth;
float blobThreshold;
int i;
char *shapesFile, *drawBaselinesFile;
char *drawBoxesFile,*plotFile,*plotOrientFile;
char *drawColorBoxesFile,*drawBlobsFile;
char *flag;
BOOLEAN doOrientation,doBaselines,doBoxes,doShapes,drawBaselines,drawBoxes;
BOOLEAN plotBaselines,plotOrientation,drawColorBoxes,drawBlobs;
BOOLEAN
noXHeightNorm,noAscenderNorm,dontOrientation,doBlobThreshold,doMaskWidth;
NormalizationDescriptor nd;

DefArg("%s","infile",&infile);
DefOption("-orientation %f","-orientation (page orientation in radians)",
        &dontOrientation,&fineAngle);
DefOption("-findOrientation","-findOrientation",&doOrientation);
DefOption("-plotOrientation %s","-plotOrientation (file top plot xgraph format image
to)",
        &plotOrientation,&plotOrientFile);
DefOption("-maskWidth %d","-maskWidth (integer half mask width)",
        &doMaskWidth,&maskWidth);
DefOption("-blobThreshold %f","-blobThreshold (float on/off threshold)",
        &doBlobThreshold,&blobThreshold);
DefOption("-drawBlobs %s","-drawBlobs (file to output image
to)",&drawBlobs,&drawBlobsFile);
DefOption("-drawBaselines %s","-drawBaselines (file to output image
to)",&drawBaselines,
        &drawBaselinesFile);
DefOption("-plotBaselines %s","-plotBaselines (file to plot xgrapgh format baselines to)",
        &plotBaselines,&plotFile);
DefOption("-drawBoxes %s","-drawBoxes (file to output image
to)",&drawBoxes,&drawBoxesFile);
DefOption("-shapeFunctions %s","-shapeFunctions (file to output shape functions to)",
        &doShapes,&shapesFile);
DefOption("-annotatedShapes %s","-annotatedShapes (file to output image to)",
        &drawColorBoxes,&drawColorBoxesFile);
DefOption("-noAscenderNorm","-noAscenderNorm",&noAscenderNorm);
DefOption("-noXHeightNorm","-noXHeightNorm",&noXHeightNorm);

i = 2;
coarseDirections = 72;
coarseSamples = 400;
fineDirections = 40;
fineSamples = 10;
maskWidth = 3;
blobThreshold = 0.01;

ScanArgs(argc,argv);
```

```
 if (dontOrientation)
  doOrientation = FALSE;

 nd.noXHeightNormalize = noXHeightNorm;
 nd.noAscenderNormalize = noAscenderNorm;

 printf("Loading %s...\n",infile);
 pict = load_pict(infile);
 if (pict->depth != 1)
  DoError("error: only depth 1 is supported\n",NULL);

 if (drawBaselines || drawBoxes)
  finalPict = new_pict(pict->width,pict->height,pict->depth);

 if (doOrientation) {
#define NUMBER_OF_ANGLES 180
#define SAMPLES_PER_ANGLE 10
#define BIN_ERROR 4
  printf("Finding coarse orientation.\n");
  coarseAngle = NewFine(pict,SAMPLES_PER_ANGLE,NUMBER_OF_ANGLES,
                 0,M_PI,NULL);
  coarseError = (M_PI-0)/NUMBER_OF_ANGLES;
  printf("Coarse angle: %f(%f)\n",coarseAngle,coarseAngle/M_PI*180);
  printf("Coarse error: %f(%f)\n",coarseError,coarseError/M_PI*180);

  mediumAngle = NewFine(pict,SAMPLES_PER_ANGLE,NUMBER_OF_ANGLES,
                   coarseAngle-BIN_ERROR*coarseError,
                   coarseAngle+BIN_ERROR*coarseError,
                   NULL);
  mediumError = 2*BIN_ERROR*coarseError/NUMBER_OF_ANGLES;
  printf("Medium angle: %f(%f)\n",mediumAngle,mediumAngle/M_PI*180);
  printf("Medium error: %f(%f)\n",mediumError,mediumError/M_PI*180);


  fineAngle = NewFine(pict,SAMPLES_PER_ANGLE,NUMBER_OF_ANGLES,
               mediumAngle-15*mediumError,mediumAngle+15*mediumError,
               plotOrientFile);
  fineError = 30*mediumError/NUMBER_OF_ANGLES;
  fineAngle = FixAngle(fineAngle);
  printf("Fine angle: %f(%f)\n",fineAngle,fineAngle/M_PI*180);
  printf("Fine error: %f(%f)\n",fineError,fineError/M_PI*180);
 }

 printf("Adjusted angle: %lf\n",fineAngle);

#ifdef foo
 printf("Finding baselines\n");
 baselines = BaseLines(pict,fineAngle,plotBaselines?plotFile:NULL);

 if (drawBaselines) {
  CopyPicture(finalPict,pict);
  DrawBaseLines(finalPict,baselines,fineAngle);
  write_pict(drawBaselinesFile,finalPict);
 }

```

```
  printf("Blobifying\n");
  newPict = Blobify(pict,maskWidth,blobThreshold);
 #endif
  printf("NewBlobify\n");
 /* newPict = NewBlobify(pict,maskWidth,blobThreshold,fineAngle); */
  newPict = Blobify(pict,maskWidth,blobThreshold);
  printf("Finding baselines\n");
  baselines = BaseLines(newPict,fineAngle,plotBaselines?plotFile:NULL);
  if (drawBaselines) {
   CopyPicture(finalPict,pict);
   DrawBaseLines(finalPict,baselines,fineAngle);
   write_pict(drawBaselinesFile,finalPict);
  }


  DrawMiddleLines(newPict,baselines,fineAngle);
  if (drawBlobs)
   write_pict(drawBlobsFile,newPict);
  printf("Finding boxes\n");
  boxList = FindBorders(newPict,fineAngle);

  if (drawBoxes) {
   CopyPicture(finalPict,pict);
   DrawBoxList(finalPict,boxList);
   write_pict(drawBoxesFile,finalPict);
  }

  if (doShapes) {
   ColorMap cmap;
   int x,y;

   if (drawColorBoxes) {
    thePict = new_pict(pict->width,pict->height,8);
    cmap = NewColorMap(6); /* black, white, and 16 colors */
    WriteColorValue(cmap,0,0,128,0);  /* Olive */
    WriteColorValue(cmap,1,0,0,0);    /* Black */
    WriteColorValue(cmap,2,255,255,255); /* White */
    WriteColorValue(cmap,3,0,0,255);  /* Blue */
    WriteColorValue(cmap,4,255,255,80); /* Yellow */
    WriteColorValue(cmap,5,128,0,0);  /* Blood */
    thePict->cmap = cmap;
    for (y=0;y<pict->height; ++y)
        for (x=0;x<pict->width; ++x)
         WritePixel(thePict,x,y,ReadPixel(pict,x,y)?0:1);
   }
   else
    thePict = NULL; /* Important */

   printf("Tracing outlines\n");
   BarBoxList(pict,boxList,baselines,shapesFile,ArgListToString(argc,argv),&nd);

   if (drawColorBoxes) {
    Dictionary dict;

    dict = ReadDictionary(shapesFile);
```

```
        LabelShapes(thePict,dict);
        write_pict(drawColorBoxesFile,thePict);
      }
    }
  }
```

Jan 11 17:07 1991 numbers.c

```
#include "stdio.h"
#include "boolean.h"
#include "pict.h"
#include "lines.h"

static float localScale;
static int localColor;
static int localX;
static int localY;
static Picture localPict;

void DrawSegment(float y1,float x1,float y2,float x2)
{
 DrawLine(localPict,irint(localX+x1*localScale),
        irint(localY+y1*localScale),
        irint(localX+x2*localScale),
        irint(localY+y2*localScale),localColor);
}

void Draw0(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(1,0,1,1);
 DrawSegment(0,0,1,0);
 DrawSegment(0,1,1,1);
}

void Draw1(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0.5,1,.5);
}

void Draw2(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(0,1,.5,1);
 DrawSegment(.5,1,.5,0);
```

```
 DrawSegment(.5,0,1,0);
 DrawSegment(1,0,1,1);
}

void Draw3(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(0,1,1,1);
 DrawSegment(1,0,1,1);
 DrawSegment(.5,0,.5,1);
}

void Draw4(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,.5,0);
 DrawSegment(0,1,1,1);
 DrawSegment(.5,0,.5,1);
}

void Draw5(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(0,0,.5,0);
 DrawSegment(.5,1,.5,0);
 DrawSegment(.5,1,1,1);
 DrawSegment(1,0,1,1);
}

void Draw6(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(0,0,1,0);
 DrawSegment(.5,1,.5,0);
 DrawSegment(.5,1,1,1);
 DrawSegment(1,0,1,1);
```

```
}

void Draw7(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(0,1,1,1);
}

void Draw8(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(0,0,1,0);
 DrawSegment(1,0,1,1);
 DrawSegment(.5,1,.5,0);
 DrawSegment(0,1,1,1);
}

void Draw9(Picture pict, int x, int y, int color,float scale)
{
 localPict = pict;
 localScale = scale;
 localColor = color;
 localX = x;
 localY = y;
 DrawSegment(0,0,0,1);
 DrawSegment(.5,0,.5,1);
 DrawSegment(0,0,.5,0);
 DrawSegment(0,1,1,1);
}

typedef void DrFct(Picture pict, int x, int y, int color, float scale);

DrFct *DrawFunctions[] = {Draw0,Draw1,Draw2,Draw3,Draw4,Draw5,Draw6,
                    Draw7,Draw8,Draw9};

void DrawNumeral(Picture pict, int x, int y, int color, float scale, int n)
{
 (*DrawFunctions[n])(pict,x,y,color,scale);
}

void DrawNumber(Picture pict, int x, int y, int color, float scale, int n)
{
 char s[100];
 char *ptr;

```

```
 sprintf(s,"%d",n);
 ptr = s;
 while (*ptr != '\0') {
  DrawNumeral(pict,x,y,color,scale,*ptr-'0');
  x += irint(scale*1.5);
  ptr++;
 }
}

#ifdef TRYMAIN
main()
{
 Picture pict;
 pict = new_pict(400,200,1);
 DrawNumber(pict,50,50,1,20,12345);
 DrawNumber(pict,50,100,1,10,67890);
 write_pict("junkfile.image",pict);
}
#endif
```

Jul 2 18:48 1991 orient.c

```
#include <stdio.h>
#include <values.h>
#include <math.h>
#include "misc.h"
#include "boolean.h"
#include "pict.h"
#include "orient.h"
#include "lines.h"


#define ABS(x) (((x)<0)?-(x):(x))

extern long random();

int RandomCoordinate(int maxValue)
{
 return (float)(random()&0xffff)*maxValue/0xffff;
}

void RandomEdgePixel(Picture pict,int *x, int *y)
{
 while (TRUE) {
  *x = RandomCoordinate(pict->width);
  *y = RandomCoordinate(pict->height);
  if (ReadPixel(pict,*x,*y))
   if (!(ReadPixel(pict,*x+1,*y) &&
         ReadPixel(pict,*x-1,*y) &&
         ReadPixel(pict,*x,*y+1) &&
         ReadPixel(pict,*x,*y-1) &&
         ReadPixel(pict,*x+1,*y+1) &&
         ReadPixel(pict,*x-1,*y-1) &&
         ReadPixel(pict,*x+1,*y-1) &&
         ReadPixel(pict,*x-1,*y+1)))
      return;
 }

}

/* #define SYMTHRESH 4 */
#define SYMTHRESH 0.17453278
BOOLEAN FindBestMin(float *distances, int coarseDirections, float step,
               float *orientation)
{
 int i,j,minIndex,min2Index;
 int orientationError;
 float minValue,min2Value;
 int maxBinError = irint(SYMTHRESH / step);

 minIndex = 0;
 minValue = distances[0];
 for (i=0;i<coarseDirections; ++i)
  if (distances[i]<minValue) {
```

```
      minValue = distances[i];
      minIndex = i;
     }
    /* Now verify that there is another minima M_PI away */

    min2Index = (minIndex+coarseDirections/4)%coarseDirections;
    min2Value = distances[min2Index];
    for (i=0,j=min2Index;i<coarseDirections/2;++i,j = (j+1)%coarseDirections)
     if (distances[j]<min2Value) {
      min2Value = distances[j];
      min2Index = j;
     }
    orientationError = ABS((min2Index-minIndex)%coarseDirections) -
     coarseDirections/2;
    orientationError = ABS(orientationError);
    if (orientationError<maxBinError) {
     *orientation = minIndex*step;
     return TRUE;
    } else {
     printf("Orientation error: %d %3.3f\n",orientationError,
           orientationError*step/M_PI/2*360);
     printf("%3.3f:%3.3f %3.3f:%3.3f\n",minIndex*step,minValue,
           min2Index*step,min2Value);
     return FALSE;
    }
   }

   float Fine(Picture pict,int fineSamples, int fineDirections,
           int coarseDirections, float coarseAngle, char *plotFile)
   {
    float coarseError;
    int x,y;
    float x2,y2;
    int i,j;
    float *counters;
    float step,angle;
    float maxAngle;
    float maxValue;
    float maxLength;
    FILE *outfile;

    counters = (float *)calloc(fineDirections,sizeof(float));
    if (counters == NULL) {
     printf("Fine: cannot allocate memory\n");
     exit(-1);
    }
   /* coarseError = 2*(SYMTHRESH+1)*2*M_PI/coarseDirections; */
    coarseError = 2*SYMTHRESH;
    step = coarseError/fineDirections;
    printf("fine: +/- %3.3f\n",fineDirections/2*step);

    maxLength = sqrt((double)(pict->width*pict->width+
                    pict->height*pict->height));
    for (i=0;i<fineSamples;++i) {
     RandomEdgePixel(pict,&x,&y);
```

```
    angle = -fineDirections/2*step+coarseAngle;
    for (j=0;j<fineDirections; ++j,angle+=step) {
     x2 = x + maxLength*cos(angle);
     y2 = y + maxLength*sin(angle);
     counters[j] += CountLine(pict,x,y,(int)x2,(int)y2);
    }
   }

   angle = -fineDirections/2*step+coarseAngle;
   maxAngle = angle;
   maxValue = counters[0];
   for (i=0;i<fineDirections; ++i,angle += step) {
  /*   printf("%3.3f: %3.3f\n",angle,counters[i]); */
    if (counters[i]>maxValue) {
     maxAngle = angle;
     maxValue = counters[i];
    }
   }

  /* Plot the orientation graph if requested */
   angle = -fineDirections/2*step+coarseAngle;
   if (angle < 0)
    angle += 2*M_PI;
   if (plotFile!=NULL) {
    printf("Opening fine orientation plot file\n");
    if ((outfile = fopen(plotFile,"a"))==NULL) {
     printf("Error opening fine orientation plot file.\n");
     exit(-1);
    }
    for (i=0;i<fineDirections; ++i, angle += step)
     fprintf(outfile,"%f %f\n",fmod(angle,2*M_PI),counters[i]);
    fprintf(outfile,"\"Fine Distances\n\n");
    fclose(outfile);
    printf("Done writing fine orientation plot file.\n");
   }


   return maxAngle;
  }

  float NewFine(Picture pict,int fineSamples, int fineDirections,
            float angleStart,float angleEnd, char *plotFile)
  {
   int x,y;
   float x2,y2;
   int i,j;
   float *counters;
   float step,angle;
   float maxAngle;
   float maxValue;
   float maxLength;
   FILE *outfile;

   counters = (float *)calloc(fineDirections,sizeof(float));
   if (counters == NULL) {
```

```
    printf("Fine: cannot allocate memory\n");
    exit(-1);
   }

   step = ABS(angleEnd - angleStart)/fineDirections;

   maxLength = sqrt((double)(pict->width*pict->width+
                    pict->height*pict->height));
   for (i=0;i<fineSamples;++i) {
    RandomEdgePixel(pict,&x,&y);
    angle = angleStart;
    for (j=0;j<fineDirections;++j) {
     angle = fmod(angle,2*M_PI);
     x2 = x + maxLength*cos(angle);
     y2 = y + maxLength*sin(angle);
     counters[j] += CountLine(pict,x,y,(int)x2,(int)y2);
     angle += step;
    }
   }

   angle = angleStart;
   maxAngle = angle;
   maxValue = counters[0];
   for (i=0;i<fineDirections;++i) {
    angle = fmod(angle,2*M_PI);
    if (counters[i]>maxValue) {
     maxAngle = angle;
     maxValue = counters[i];
    }
    angle += step;
   }
   printf("Orientation is at %f(%f)\n",maxAngle,maxAngle/2/M_PI*360);

  /* Plot the orientation graph if requested */
   if (plotFile) {
    printf("Opening fine orientation plot file\n");
    if ((outfile = fopen(plotFile,"w"))==NULL) {
     printf("Error opening fine orientation plot file.\n");
     exit(-1);
    }
    angle = angleStart;
    for (i=0;i<fineDirections;++i) {
     angle = fmod(angle,2*M_PI);
     fprintf(outfile,"%f %f\n",angle,counters[i]);
     angle += step;
    }
    fprintf(outfile,"\"Fine Distances\n\n");
    fclose(outfile);
    printf("Done writing fine orientation plot file.\n");
   }
   return maxAngle;
  }
```

Jan 15 15:22 1991 overlay.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"

main(argc, argv)
int argc;
char *argv[];
{
 char *inFile1,*inFile2,*outFile;
 Picture pict1,pict2,finalPict;
 ColorMap cmap;
 int x,y;

 if (argc != 4)
 {
  printf("\nUsage: %s infile1 infile2 outfile\n\n",
         argv[0]);
  exit(0);
 }

 inFile1 = argv[1];    /* get args */
 inFile2 = argv[2];
 outFile = argv[3];

 pict1 = load_pict(inFile1);
 pict2 = load_pict(inFile2);
 if ((pict1->depth != 1) || (pict2->depth != 1))
  DoError("overlay: only depth 1 supported.\n",NULL);
 if ((pict1->width != pict2->width)||(pict1->height != pict2->height))
  DoError("overlay: images must be the same size\n",NULL);

 finalPict = new_pict(pict1->width,pict1->height,8);
 cmap = NewColorMap(3);
 WriteColorValue(cmap,0,0,0,0);     /* Black */
 WriteColorValue(cmap,1,0,128,0);   /* Olive */
 WriteColorValue(cmap,2,0,255,0);   /* Green */
 finalPict->cmap = cmap;

 for (y=0;y<pict1->height;++y)
  for (x=0;x<pict1->width;++x)
   if (ReadPixel(pict1,x,y))
       WritePixel(finalPict,x,y,2);
   else if (ReadPixel(pict2,x,y))
       WritePixel(finalPict,x,y,1);

 write_pict(outFile,finalPict);
}
```

Jul 1 13:45 1991 pagestats.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"

#define UP 0
#define DOWN 1
typedef int Direction;

extern Picture thePict;

void StoreRawOutlinePair(Dictionary dict, int dictEntry,
                         Box box,int *bothX,int *topY,int *baseY,
                         int numberOfLegs)
{
  RawOutlinePair temp;
  int i;
  int *xCursor,*topCursor,*bottomCursor;

  temp = (RawOutlinePair)calloc(1,sizeof(RawOutlinePairBody));
  if (temp == NULL)
    DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

  temp->box = box;
  temp->numberOfLegs = numberOfLegs;

  temp->x = (int *)calloc(temp->numberOfLegs,sizeof(int));
  temp->top = (int *)calloc(temp->numberOfLegs,sizeof(int));
  temp->bottom = (int *)calloc(temp->numberOfLegs,sizeof(int));
  if ((temp->x == NULL) ||
      (temp->top == NULL) ||
      (temp->bottom == NULL))
    DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

  xCursor = temp->x;
  topCursor = temp->top;
  bottomCursor = temp->bottom;

  for (i=0;i<numberOfLegs;++i) {
    *xCursor++ = *bothX++;
    *topCursor++ = *topY++;
    *bottomCursor++ = *baseY++;
  }
  *(dict->rawOutlines+dictEntry) = temp;
}

void StoreOutlinePair(Dictionary dict, int dictEntry,
                      int middleLine,int fontXHeight)
{
```

```
    RawOutlinePair raw;
    OutlinePair temp;
    int i,numberOfLegs;
    int y;
    int offset;
    int *xSCursor,*topSCursor,*bottomSCursor;
    float *xDCursor,*topDCursor,*bottomDCursor;
    float *xCursor,*topCursor,*bottomCursor;
    int left,right;
    float foffset;

    raw = *(dict->rawOutlines+dictEntry);

    temp = (OutlinePair)calloc(1,sizeof(OutlinePairBody));
    if (temp == NULL)
     DoError("StoreOutlinePair: cannot allocate space\n",NULL);

    temp->x = (float *)calloc(raw->numberOfLegs,sizeof(float));
    temp->top = (float *)calloc(raw->numberOfLegs,sizeof(float));
    temp->bottom = (float *)calloc(raw->numberOfLegs,sizeof(float));
    if ((temp->x == NULL) ||
      (temp->top == NULL) ||
      (temp->bottom == NULL))
     DoError("StoreOutlinePair: cannot allocate space\n",NULL);

    temp->box = raw->box;
    temp->blackoutHeight = 0;
    temp->numberOfLegs = raw->numberOfLegs;
    offset = temp->offset = *(raw->x);
    temp->width = *(raw->x+raw->numberOfLegs-1) - temp->offset;

    xDCursor = temp->x;
    topDCursor = temp->top;
    bottomDCursor = temp->bottom;
    xSCursor = raw->x;
    topSCursor = raw->top;
    bottomSCursor = raw->bottom;

    numberOfLegs = raw->numberOfLegs;
    for (i=0;i<numberOfLegs;++i) {
     *xDCursor++ = (float)(*xSCursor++ - offset)/fontXHeight;
     y = middleLine - *topSCursor++;
     if (y<0)
      y = 0;
     *topDCursor++ = (float)y / fontXHeight;
     y = *bottomSCursor++ - middleLine;
     if (y<0)
      y = 0;
     *bottomDCursor++ = (float)y / fontXHeight;
    }

   /* Now try to remove parts of the contour on to the left and right of the
    * word shape that are at height 0 */

    topDCursor = temp->top;
```

```
  bottomDCursor = temp->bottom;
  for (i=0;i<numberOfLegs;++i) {
   if ((*topDCursor++ != 0)||(*bottomDCursor++!=0))
    break;
  }
  left = i;

  topDCursor = temp->top+numberOfLegs-1;
  bottomDCursor = temp->bottom+numberOfLegs-1;
  for (i=numberOfLegs-1;i>=0;--i) {
   if ((*topDCursor-- != 0)||(*bottomDCursor-- !=0))
    break;
  }
  right = i+1;

  xDCursor = temp->x;
  topDCursor = temp->top;
  bottomDCursor = temp->bottom;
  xCursor = temp->x+left;
  topCursor = temp->top+left;
  bottomCursor = temp->bottom+left;
  foffset = *xSCursor;
  for (i=left;i<right;++i) {
   *xDCursor++ = *xCursor++ - foffset;
   *topDCursor++ = *topCursor++;
   *bottomDCursor++ = *bottomCursor++;
  }
  temp->numberOfLegs = right-left;

  *(dict->outlines+dictEntry) = temp;
}

static int lineSpacing;
int OrderOutlinePair(OutlinePair *o1,OutlinePair *o2)
{
 int yDistance;
 int xDistance;
 yDistance = (*o1)->box->pageY - (*o2)->box->pageY;
 if (yDistance<lineSpacing && yDistance > -lineSpacing) {
  xDistance = (*o1)->box->pageX - (*o2)->box->pageX;
  return xDistance;
 }
 return yDistance;
}

void SortDictionary(Dictionary dict)
{
 lineSpacing = 20;
 qsort(dict->rawOutlines,dict->numberOfEntries,sizeof(RawOutlinePair),
       OrderOutlinePair);
}

#define HIST_SIZE 100
void Histogram(int *data,int dataLength, int offset, int *histogram)
{
```

```
 int i,bin;
 for (i=0;i<dataLength; ++i) {
  bin = *data-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
  data++;
 }
}

void HistogramPeaks(int *data,int dataLength, int offset, int *histogram)
{
 int i,bin;
 Direction direction;

 if (*(data+1) < *data)
  direction = UP;
 else {
  bin = *data-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
  direction = DOWN;
 }
 ++data;

 for (i=1;i<dataLength-1;++i) {
  if ((direction == UP) &&
      (*data < *(data+1))) {
   /* *data is a peak */
   bin = *data-offset;
   if ((bin>=0)&&(bin<HIST_SIZE))
       histogram[bin]++;
   direction = DOWN;
  }
  else if ((direction == DOWN) &&
        (*data > *(data+1))) {
   /* *data is a valley */
   direction = UP;
  }
  ++data;
 } /* for i */
}

void HistogramValleys(int *data,int dataLength, int offset, int *histogram)
{
 int i,bin;
 Direction direction;

 if (*(data+1) > *data)
  direction = UP;
 else {
  bin = *data-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
  direction = DOWN;
 }
```

```
    ++data;

    for (i=1;i<dataLength-1;++i) {
     if ((direction == UP) &&
          (*data > *(data+1))) {
      /* *data is a peak */
      bin = *data-offset;
      if ((bin>=0)&&(bin<HIST_SIZE))
          histogram[bin]++;
      direction = DOWN;
     }
     else if ((direction == DOWN) &&
             (*data < *(data+1))) {
      /* *data is a valley */
      direction = UP;
     }
     ++data;
    } /* for i */
}

int MaxBin(int *histogram)
{
 int i;
 int maxValue;
 int maxIndex;

 maxValue = histogram[0];
 maxIndex = 0;
 for (i=0;i<HIST_SIZE;++i)
  if (histogram[i]>maxValue) {
   maxValue = histogram[i];
   maxIndex = i;
  }
 return maxIndex;
}

void PostProcess(Dictionary dict)
{
 int index;
 int temp;
 int i,startIndex,firstY,minY,endIndex,shape;
 int tops[HIST_SIZE];
 int bottoms[HIST_SIZE];
 int middleLine,topLine,bottomLine;
 int fontXHeight;
 RawOutlinePair thisShape;

 SortDictionary(dict);

 index = 0;
#ifdef foo
 malloc_verify();
#endif
 while (index < dict->numberOfEntries) {
  startIndex = index;
```

```
  firstY = (*(dict->rawOutlines+index))->box->pageY;
  minY = firstY;
  while ( (*(dict->rawOutlines+index))->box->pageY - firstY < 20 &&
         (*(dict->rawOutlines+index))->box->pageY - firstY > -20) {
   if ( minY > ( (*(dict->rawOutlines+index))->box->pageY ))
      minY = (*(dict->rawOutlines+index))->box->pageY;
   ++index;
   if (index == dict->numberOfEntries)
      break;
  }
  endIndex = index;

#ifdef foo
  malloc_verify();
#endif

  /* shapes from start index through endindex are all on */
  /* the same text line */
  /* minY has the top of the highest box on the line. */

  /* Find the base and toplines by taking the mode of the heights of the
   * valleys of the bottom contours and the peaks of the top contours */
  for (i=0;i<HIST_SIZE;i++) {
   tops[i]=0;
   bottoms[i]=0;
  }
  for (shape=startIndex;shape<endIndex;++shape) {
   thisShape = *(dict->rawOutlines+shape);
   Histogram(thisShape->top,thisShape->numberOfLegs,minY,tops);
   Histogram(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);

#ifdef foo
   HistogramPeaks(thisShape->top,thisShape->numberOfLegs,minY,tops);
   HistogramValleys(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);
#endif
  }
  topLine = MaxBin(tops)+minY;
  bottomLine = MaxBin(bottoms)+minY;

  if (thePict) {
   int maxLength;
   int halfWidth;
   int x,y;
   float x2,x3,y2,y3;
   float angle;

   angle = (*(dict->rawOutlines))->box->angle;
   maxLength = thePict->width+thePict->height;
   halfWidth = thePict->width / 2;
   x = topLine * -sin(angle) + halfWidth * cos(angle);
   y = topLine * cos(angle) + halfWidth * sin(angle);
   x2 = x+maxLength*cos(angle);
   y2 = y+maxLength*sin(angle);
   x3 = x-maxLength*cos(angle);
   y3 = y-maxLength*sin(angle);
```

```
            DrawLine(thePict,x,y,(int)x2,(int)y2,5);
            DrawLine(thePict,x,y,(int)x3,(int)y3,5);

            x = bottomLine * -sin(angle) + halfWidth * cos(angle);
            y = bottomLine * cos(angle) + halfWidth * sin(angle);
            x2 = x+maxLength*cos(angle);
            y2 = y+maxLength*sin(angle);
            x3 = x-maxLength*cos(angle);
            y3 = y-maxLength*sin(angle);
            DrawLine(thePict,x,y,(int)x2,(int)y2,5);
            DrawLine(thePict,x,y,(int)x3,(int)y3,5);

        }

    #ifdef foo
      malloc_verify();
    #endif

      middleLine = (bottomLine+topLine)/2;
      fontXHeight = bottomLine-topLine;
      /* Clip and normalize the contours */
      for (shape=startIndex;shape<endIndex;++shape)
        StoreOutlinePair(dict,shape,middleLine,fontXHeight);
     } /* Do another line of text */
    }
```

Jul 1 13:46 1991 postproc.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "types.h"
#include "error.h"
#include "pict.h"
#include "dict.h"

#define UP 0
#define DOWN 1
typedef int Direction;

extern Picture thePict;

void StoreRawOutlinePair(Dictionary dict, int dictEntry,
                         Box box,int *bothX,int *topY, int *baseY,
                         int numberOfLegs)
{
 RawOutlinePair temp;
 int i;
 int *xCursor,*topCursor,*bottomCursor;

 temp = (RawOutlinePair)calloc(1,sizeof(RawOutlinePairBody));
 if (temp = = NULL)
  DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

 temp->box = box;
 temp->numberOfLegs = numberOfLegs;

 temp->x = (int *)calloc(temp->numberOfLegs,sizeof(int));
 temp->top = (int *)calloc(temp->numberOfLegs,sizeof(int));
 temp->bottom = (int *)calloc(temp->numberOfLegs,sizeof(int));
 if ((temp->x = = NULL) ||
   (temp->top = = NULL) ||
   (temp->bottom = = NULL))
  DoError("StoreRawOutlinePair: cannot allocate space\n",NULL);

 xCursor = temp->x;
 topCursor = temp->top;
 bottomCursor = temp->bottom;

 for (i=0;i<numberOfLegs; + +i) {
  *xCursor+ + = *bothX+ +;
  *topCursor+ + = *topY+ +;
  *bottomCursor+ + = *baseY+ +;
 }
 *(dict->rawOutlines+dictEntry) = temp;
}

void StoreOutlinePair(Dictionary dict, int dictEntry,
                  int middleLine,int fontXHeight)
{
```

```
    RawOutlinePair raw;
    OutlinePair temp;
    int i,numberOfLegs;
    int y;
    int offset;
    int *xSCursor,*topSCursor,*bottomSCursor;
    float *xDCursor,*topDCursor,*bottomDCursor;
    float *xCursor,*topCursor,*bottomCursor;
    int left,right;
    float foffset;

    raw = *(dict->rawOutlines+dictEntry);

    temp = (OutlinePair)calloc(1,sizeof(OutlinePairBody));
    if (temp == NULL)
     DoError("StoreOutlinePair: cannot allocate space\n",NULL);

    temp->x = (float *)calloc(raw->numberOfLegs,sizeof(float));
    temp->top = (float *)calloc(raw->numberOfLegs,sizeof(float));
    temp->bottom = (float *)calloc(raw->numberOfLegs,sizeof(float));
    if ((temp->x == NULL) ||
      (temp->top == NULL) ||
      (temp->bottom == NULL))
     DoError("StoreOutlinePair: cannot allocate space\n",NULL);

    temp->box = raw->box;
    temp->blackoutHeight = 0;
    temp->numberOfLegs = raw->numberOfLegs;
    offset = temp->offset = *(raw->x);
    temp->width = *(raw->x+raw->numberOfLegs-1) - temp->offset;

    xDCursor = temp->x;
    topDCursor = temp->top;
    bottomDCursor = temp->bottom;
    xSCursor = raw->x;
    topSCursor = raw->top;
    bottomSCursor = raw->bottom;

    numberOfLegs = raw->numberOfLegs;
    for (i=0;i<numberOfLegs;++i) {
     *xDCursor++ = (float)(*xSCursor++ - offset)/fontXHeight;
     y = middleLine - *topSCursor++;
     if (y<0)
      y = 0;
     *topDCursor++ = (float)y / fontXHeight;
     y = *bottomSCursor++ - middleLine;
     if (y<0)
      y = 0;
     *bottomDCursor++ = (float)y / fontXHeight;
    }

   /* Now try to remove parts of the contour on to the left and right of the
    * word shape that are at height 0 */

    topDCursor = temp->top;
```

```
  bottomDCursor = temp->bottom;
  for (i=0;i<numberOfLegs;++i) {
   if ((*topDCursor++ != 0)||(*bottomDCursor++!=0))
    break;
  }
  left = i;

  topDCursor = temp->top+numberOfLegs-1;
  bottomDCursor = temp->bottom+numberOfLegs-1;
  for (i=numberOfLegs-1;i>=0;--i) {
   if ((*topDCursor-- != 0)||(*bottomDCursor-- !=0))
    break;
  }
  right = i+1;

  xDCursor = temp->x;
  topDCursor = temp->top;
  bottomDCursor = temp->bottom;
  xCursor = temp->x+left;
  topCursor = temp->top+left;
  bottomCursor = temp->bottom+left;
  foffset = *xSCursor;
  for (i=left;i<right;++i) {
   *xDCursor++ = *xCursor++ - foffset;
   *topDCursor++ = *topCursor++;
   *bottomDCursor++ = *bottomCursor++;
  }
  temp->numberOfLegs = right-left;

  *(dict->outlines+dictEntry) = temp;
 }

static int lineSpacing;
int OrderOutlinePair(OutlinePair *o1,OutlinePair *o2)
{
 int yDistance;
 int xDistance;
 yDistance = (*o1)->box->pageY - (*o2)->box->pageY;
 if (yDistance<lineSpacing && yDistance > -lineSpacing) {
  xDistance = (*o1)->box->pageX - (*o2)->box->pageX;
  return xDistance;
 }
 return yDistance;
}

void SortDictionary(Dictionary dict)
{
 lineSpacing = 20;
 qsort(dict->rawOutlines,dict->numberOfEntries,sizeof(RawOutlinePair),
       OrderOutlinePair);
}

#define HIST_SIZE 100
void HistogramMax(int *data,int dataLength,int offset,int sign,int *histogram)
{
```

```
 int i,bin;

 if (sign>0) {
  int maxValue;
  maxValue = *data;
  for (i=0;i<dataLength;++i)
   if (data[i]>maxValue)
        maxValue = data[i];
  bin = maxValue-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
 }
 else {
  int minValue;
  minValue = *data;
  for (i=0;i<dataLength;++i)
   if (data[i]<minValue)
        minValue = data[i];
  bin = minValue-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
 }
}

void Histogram(int *data,int dataLength, int offset, int *histogram)
{
 int i,bin;

 for (i=0;i<dataLength;++i) {
  bin = *data-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
  data++;
 }
}

void HistogramPeaks(int *data,int dataLength, int offset, int *histogram)
{
 int i,bin;
 Direction direction;

 if (*(data+1) < *data)
  direction = UP;
 else {
  bin = *data-offset;
  if ((bin>=0)&&(bin<HIST_SIZE))
   histogram[bin]++;
  direction = DOWN;
 }
 ++data;

 for (i=1;i<dataLength-1;++i) {
  if ((direction == UP) &&
       (*data < *(data+1))) {
   /* *data is a peak */
```

```
      bin = *data-offset;
      if ((bin>=0)&&(bin<HIST_SIZE))
         histogram[bin]++;
      direction = DOWN;
     }
     else if ((direction == DOWN) &&
            (*data > *(data+1))) {
      /* *data is a valley */
      direction = UP;
     }
     ++data;
    } /* for i */
   }

   void HistogramValleys(int *data,int dataLength, int offset, int *histogram)
   {
    int i,bin;
    Direction direction;

    if (*(data+1) > *data)
     direction = UP;
    else {
     bin = *data-offset;
     if ((bin>=0)&&(bin<HIST_SIZE))
      histogram[bin]++;
     direction = DOWN;
    }
    ++data;

    for (i=1;i<dataLength-1;++i) {
     if ((direction == UP) &&
          (*data > *(data+1))) {
      /* *data is a peak */
      bin = *data-offset;
      if ((bin>=0)&&(bin<HIST_SIZE))
          histogram[bin]++;
      direction = DOWN;
     }
     else if ((direction == DOWN) &&
            (*data < *(data+1))) {
      /* *data is a valley */
      direction = UP;
     }
     ++data;
    } /* for i */
   }

   int MaxBin(int *histogram)
   {
    int i;
    int maxValue;
    int maxIndex;

    maxValue = histogram[0];
    maxIndex = 0;
```

```
    for (i=0;i<HIST_SIZE; ++i)
     if (histogram[i]>maxValue) {
      maxValue = histogram[i];
      maxIndex = i;
     }
    return maxIndex;
   }

   void PostProcess(Dictionary dict)
   {
    int index;
    int temp;
    int i,startIndex,firstY,minY,endIndex,shape;
    int tops[HIST_SIZE];
    int bottoms[HIST_SIZE];
    int middleLine,topLine,bottomLine;
    int fontXHeight;
    RawOutlinePair thisShape;

    SortDictionary(dict);

    index = 0;
   #ifdef foo
    malloc_verify();
   #endif
    while (index < dict->numberOfEntries) {
     startIndex = index;
     firstY = (*(dict->rawOutlines+index))->box->pageY;
     minY = firstY;
     while ( (*(dict->rawOutlines+index))->box->pageY - firstY < 20 &&
            (*(dict->rawOutlines+index))->box->pageY - firstY > -20) {
      if ( minY > ( (*(dict->rawOutlines+index))->box->pageY ))
          minY = (*(dict->rawOutlines+index))->box->pageY;
      ++index;
      if (index == dict->numberOfEntries)
          break;
     }
     endIndex = index;

   #ifdef foo
     malloc_verify();
   #endif

     /* shapes from start index through endindex are all on */
     /* the same text line */
     /* minY has the top of the highest box on the line. */

     /* Find the base and toplines by taking the mode of the heights of the
      * valleys of the bottom contours and the peaks of the top contours */
     for (i=0;i<HIST_SIZE;i++) {
      tops[i]=0;
      bottoms[i]=0;
     }
     for (shape=startIndex;shape<endIndex; ++shape) {
      thisShape = *(dict->rawOutlines+shape);
```

```
      Histogram(thisShape->top,thisShape->numberOfLegs,minY,tops);
      Histogram(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);

#ifdef foo
      HistogramPeaks(thisShape->top,thisShape->numberOfLegs,minY,tops);
      HistogramValleys(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);
#endif
    }
    topLine = MaxBin(tops)+minY;
    bottomLine = MaxBin(bottoms)+minY;

    if (thePict) {
     int maxLength;
     int halfWidth;
     int x,y;
     float x2,x3,y2,y3;
     float angle;

     angle = (*(dict->rawOutlines))->box->angle;
     maxLength = thePict->width+thePict->height;
     halfWidth = thePict->width / 2;
     x = topLine * -sin(angle) + halfWidth * cos(angle);
     y = topLine * cos(angle) + halfWidth * sin(angle);
     x2 = x+maxLength*cos(angle);
     y2 = y+maxLength*sin(angle);
     x3 = x-maxLength*cos(angle);
     y3 = y-maxLength*sin(angle);
     DrawLine(thePict,x,y,(int)x2,(int)y2,5);
     DrawLine(thePict,x,y,(int)x3,(int)y3,5);

     x = bottomLine * -sin(angle) + halfWidth * cos(angle);
     y = bottomLine * cos(angle) + halfWidth * sin(angle);
     x2 = x+maxLength*cos(angle);
     y2 = y+maxLength*sin(angle);
     x3 = x-maxLength*cos(angle);
     y3 = y-maxLength*sin(angle);
     DrawLine(thePict,x,y,(int)x2,(int)y2,5);
     DrawLine(thePict,x,y,(int)x3,(int)y3,5);

    }

#ifdef foo
    malloc_verify();
#endif

    middleLine = (bottomLine+topLine)/2;
    fontXHeight = bottomLine-topLine;
    /* Clip and normalize the contours */
    for (shape=startIndex;shape<endIndex;++shape)
     StoreOutlinePair(dict,shape,middleLine,fontXHeight);
  } /* Do another line of text */
}

void PageStatistics(Dictionary dict,char *fileName)
/* WARNING - this must be run before PostProcess since PostProcess changes the raw
```

```
 * shape data. */
{
 int index;
 int temp;
 int i,startIndex,firstY,minY,endIndex,shape;
 int tops[HIST_SIZE];
 int bottoms[HIST_SIZE];
 int ascenders[HIST_SIZE];
 int descenders[HIST_SIZE];
 int middleLine,topLine,bottomLine,ascenderLine,descenderLine;
 int ascenderHeight,descenderHeight,lineNumber;
 int fontXHeight;
 RawOutlinePair thisShape;
 FILE *fp;

 if ((fp=fopen(fileName,"w"))==NULL)
  DoError("PageStatistics: error opening output file %s.\n",fileName);

 SortDictionary(dict);

 index = 0;
#ifdef foo
 malloc_verify();
#endif
 lineNumber = 0;
 while (index < dict->numberOfEntries) {
  startIndex = index;
  firstY = (*(dict->rawOutlines+index))->box->pageY;
  minY = firstY;
  while ( (*(dict->rawOutlines+index))->box->pageY - firstY < 20 &&
         (*(dict->rawOutlines+index))->box->pageY - firstY > -20) {
   if ( minY > ( (*(dict->rawOutlines+index))->box->pageY ))
       minY = (*(dict->rawOutlines+index))->box->pageY;
    ++index;
   if (index == dict->numberOfEntries)
       break;
  }
  endIndex = index;

#ifdef foo
  malloc_verify();
#endif

  /* shapes from start index through endindex are all on */
  /* the same text line */
  /* minY has the top of the highest box on the line. */

  /* Find the base and toplines by taking the mode of the heights of the
   * valleys of the bottom contours and the peaks of the top contours */
  for (i=0;i<HIST_SIZE;i++) {
   tops[i]=0;
   bottoms[i]=0;
   ascenders[i]=0;
   descenders[i]=0;
  }
```

```
    for (shape=startIndex;shape<endIndex;++shape) {
     thisShape = *(dict->rawOutlines+shape);
     Histogram(thisShape->top,thisShape->numberOfLegs,minY,tops);
     Histogram(thisShape->bottom,thisShape->numberOfLegs,minY,bottoms);

     HistogramMax(thisShape->top,thisShape->numberOfLegs,minY,-1,ascenders);
     HistogramMax(thisShape->bottom,thisShape->numberOfLegs,minY,1,descenders);
    }
    topLine = MaxBin(tops)+minY;
    bottomLine = MaxBin(bottoms)+minY;
    ascenderLine = MaxBin(ascenders)+minY;
    descenderLine = MaxBin(descenders)+minY;

  #ifdef foo
    malloc_verify();
  #endif

    middleLine = (bottomLine+topLine)/2;
    fontXHeight = bottomLine-topLine;

    ascenderHeight = bottomLine-ascenderLine;
    descenderHeight = descenderLine-bottomLine;
    fprintf(fp,"%d: %d %d %d
  %2.6f\n",lineNumber,fontXHeight,ascenderHeight,descenderHeight,
          (float)ascenderHeight/(float)fontXHeight);
    ++lineNumber;
   } /* Do another line of text */
   fclose(fp);
  }
```

Jul 10 13:17 1991 testFine.c

```
#include <stdio.h>
#include <math.h>
#include "boolean.h"
#include "pict.h"
#include "lines.h"

#define ABS(x) (((x)<0)?(-(x)):(x))

extern long random();

int RandomCoordinate(int maxValue)
{
 return (float)(random()&0xffff)*maxValue/0xffff;
}

void RandomEdgePixel(Picture pict,int *x, int *y)
{
 while (TRUE) {
  *x = RandomCoordinate(pict->width);
  *y = RandomCoordinate(pict->height);
  if (ReadPixel(pict,*x,*y))
   if (!(ReadPixel(pict,*x+1,*y) &&
         ReadPixel(pict,*x-1,*y) &&
         ReadPixel(pict,*x,*y+1) &&
         ReadPixel(pict,*x,*y-1) &&
         ReadPixel(pict,*x+1,*y+1) &&
         ReadPixel(pict,*x-1,*y-1) &&
         ReadPixel(pict,*x+1,*y-1) &&
         ReadPixel(pict,*x-1,*y+1)))
     return;
 }

}

float Fine(Picture pict,int fineSamples, int fineDirections,
          float angleStart,float angleEnd, char *plotFile)
{
 int x,y;
 float x2,y2;
 int i,j;
 float *counters;
 float step,angle;
 float maxAngle;
 float maxValue;
 float maxLength;
 FILE *outfile;

 counters = (float *)calloc(fineDirections,sizeof(float));
 if (counters == NULL) {
  printf("Fine: cannot allocate memory\n");
  exit(-1);
 }
```

```
  step = ABS(angleEnd - angleStart)/fineDirections;

  maxLength = sqrt((double)(pict->width*pict->width+
                    pict->height*pict->height));
  for (i=0;i<fineSamples; ++i) {
   RandomEdgePixel(pict,&x,&y);
   angle = angleStart;
   for (j=0;j<fineDirections; ++j) {
    angle = fmod(angle,2*M_PI);
    x2 = x + maxLength*cos(angle);
    y2 = y + maxLength*sin(angle);
    counters[j] += CountLine(pict,x,y,(int)x2,(int)y2);
    angle += step;
   }
  }

  angle = angleStart;
  maxAngle = angle;
  maxValue = counters[0];
  for (i=0;i<fineDirections; ++i) {
   angle = fmod(angle,2*M_PI);
   if (counters[i]>maxValue) {
    maxAngle = angle;
    maxValue = counters[i];
   }
   angle += step;
  }
  printf("Orientation is at %f(%f)\n",maxAngle,maxAngle/2/M_PI*360);

 /* Plot the orientation graph if requested */
  printf("Opening fine orientation plot file\n");
  if ((outfile = fopen(plotFile,"w"))==NULL) {
   printf("Error opening fine orientation plot file.\n");
   exit(-1);
  }
  angle = angleStart;
  for (i=0;i<fineDirections; ++i) {
   angle = fmod(angle,2*M_PI);
   fprintf(outfile,"%f %f\n",angle,counters[i]);
   angle += step;
  }
  fprintf(outfile,"\"Fine Distances\n\n");
  fclose(outfile);
   printf("Done writing fine orientation plot file.\n");
  return maxAngle;
 }

 main(argc, argv)
 int argc;
 char *argv[];
 {
  char *inFileName,*coarseOutFileName,*fineOutFileName,*fine2OutFileName;
  int fineDirections,fineSamples;
  float coarseAngle,fineAngle,fineAngle2;
```

```
float firstSpacing,secondSpacing,thirdSpacing;
Picture pict;

if (argc != 7)
{
 printf("\nUsage: %s infile coarsePlotFile finePlotFile\n",argv[0]);
 printf("     finerPlotFile #directions #samples\n\n");

 exit(0);
}

inFileName = argv[1];    /* get args */
coarseOutFileName = argv[2];
fineOutFileName = argv[3];
fine2OutFileName = argv[4];
fineDirections = atoi(argv[5]);
fineSamples = atoi(argv[6]);

pict = load_pict(inFileName);
coarseAngle = Fine(pict,fineSamples,fineDirections,
              0,M_PI,coarseOutFileName);
firstSpacing = (M_PI-0)/fineDirections;
printf("Coarse angle: %f(%f)\n",coarseAngle,coarseAngle/M_PI*180);
printf("Coarse spacing: %f(%f)\n",firstSpacing,firstSpacing/M_PI*180);

fineAngle = Fine(pict,fineSamples,fineDirections,
             coarseAngle-4*firstSpacing,coarseAngle+4*firstSpacing,
             fineOutFileName);
secondSpacing = 8*firstSpacing/fineDirections;
printf("Fine angle: %f(%f)\n",fineAngle,fineAngle/M_PI*180);
printf("Fine spacing: %f(%f)\n",secondSpacing,secondSpacing/M_PI*180);

fineAngle2 = Fine(pict,fineSamples,fineDirections,
             fineAngle-15*secondSpacing,fineAngle+15*secondSpacing,
             fine2OutFileName);
thirdSpacing = 30*secondSpacing/fineDirections;
printf("Finer angle: %f(%f)\n",fineAngle2,fineAngle2/M_PI*180);
printf("Finer spacing: %f(%f)\n",thirdSpacing,thirdSpacing/M_PI*180);
}
```

Aug 15 06:32 1991 types.c

```
#include "stdio.h"
#include "mylib.h"
#include "types.h"
#include "error.h"

Box MakeBox(int x,int y,int width,int height,double angle)
{
 Box temp;
 temp = (Box)calloc(1,sizeof(BoxBody));
 if (temp==NULL)
  DoError("MakeBox: out of memory\n",NULL);
 temp->x = x;
 temp->y = y;
 temp->width = width;
 temp->height = height;
 temp->angle = angle;
 return temp;
}

Point MakePoint(int x,int y)
{
 Point temp;
 temp = (Point)calloc(1,sizeof(PointBody));
 if (temp==NULL)
  DoError("MakePoint: out of memory\n",NULL);
 temp->x = x;
 temp->y = y;
 return temp;
}
```

We claim:

1. A method for electronically processing an electronic document image without first decoding the electronic document image, comprising:

segmenting the document image into word image units without decoding the document image;

deriving a word shape representation for each of a plurality of said word image units without decoding any characters making up the plurality of word image units, thereby deriving a plurality of said word shape representations;

comparing said word shape representations to at least one other word shape representation to identify significant word image units from amongst said plurality of word image units; and

creating an abbreviated document image that is smaller than the electronic document image based on said identified significant word image units, said abbreviated document image including a plurality of said identified significant word image units.

2. The method of claim 1 wherein said step of comparing includes classifying said word image units according to frequency of occurrence based on comparing said word shape representations with each other.

3. The method of claim 1 wherein said step of comparing includes classifying said word image units according to location within the document image.

4. The method of claim 1 wherein said step of deriving a word shape representation includes utilization of at least one of an image unit shape dimension, font, typeface, number of ascender elements, number of descender elements, pixel density, pixel cross-sectional characteristic, the location of word image units with respect to neighboring word image units, vertical position, horizontal interimage unit spacing, and contour characteristic of said word image units.

5. The method of claim 1, wherein said comparing step includes comparing said word shape representations with each other.

6. The method of claim 1, wherein said comparing step includes comparing said word shape representations with at least one predetermined word shape representation.

7. The method of claim 1, wherein said comparing step includes comparing said word shape representations with at least one user-selected word shape representation.

8. A method of excerpting significant information from an undecoded document image without decoding the document image, comprising:

segmenting the document image into word image units without decoding the document image;

deriving a word shape representation for each of a plurality of said word image units without decoding any characters making up said plurality of word image units, thereby deriving a plurality of said word shape representations;

comparing said word shape representations to at least one other word shape representation to identify significant word image units from amongst said word image units; and

outputting a plurality of said identified significant word image units for further processing.

9. The method of claim 8 wherein said step of outputting a plurality of identified significant image units comprises generating a document index based on said significant identified word image units.

10. The method of claim 8 wherein said step of outputting a plurality of identified significant image units comprises producing a speech synthesized output corresponding to said identified significant word image units.

11. The method of claim 8 wherein said step of outputting a plurality of identified significant word image units comprises producing said identified significant word image units in printed Braille format.

12. The method of claim 8 wherein said step of outputting said a plurality of identified significant word image units comprises generating a document summary from said identified significant word image units.

13. A method for electronically processing an undecoded document image containing word text, comprising:

segmenting the document image into word image units without decoding the document image;

deriving a word shape representation for each of a plurality of said word image units without decoding any characters making up said plurality of word image units, thereby deriving a plurality of said word shape representations;

comparing said word shape representations to at least one other word shape representation to identify significant word image units from amongst said plurality of word image units;

forming phrase image units based on a plurality of said identified significant word image units, said phrase image units each incorporating one of said identified significant word image units and adjacent word image units linked in reading order sequence; and

outputting said phrase image units.

14. An apparatus for automatically summarizing the information content of an undecoded document image without decoding the document image, comprising:

means for segmenting the document image into word image units without decoding the document image;

means for deriving a word shape representation for each of a plurality of said word image units without decoding any characters making up said plurality of word image units, thereby deriving a plurality of said word shape representations;

means for comparing said word shape representations to at least one other word shade representation to identify significant word image units from amongst said plurality of word image units; and

means for creating a supplemental document image based on said identified significant word image units.

15. The apparatus of claim 14 wherein said means for segmenting the document image, said means for deriving a word shape representation, said means for comparing, said means for creating a supplemental document image comprise a programmed digital computer.

16. The apparatus of claim 15 further comprising scanning means for scanning an original document to produce said document image, said scanning means being incorporated in a document copier machine which produces printed

document copies; and means for controlling said document copier machine to produce a printed document copy of said supplemental document image.

17. The apparatus of claim **15** further comprising scanning means for scanning an original document to produce said document image, said scanning means being incorporated in a reading machine for the blind having means for communicating data to the user; and means for controlling said reading machine communication means to communicate the contents of said supplemental document image.

18. The apparatus of claim **17** wherein said communicating means comprises a printer for producing document copies in Braille format.

19. The apparatus of claim **17** wherein said communicating means comprises a speech synthesizer for producing synthesized speech output corresponding to said supplemental document image.

20. The apparatus of claim **17** wherein said reading machine includes operator responsive means for accessing the scanned document or a selected portion thereof corresponding to a supplemental document image following communication of the supplemental document image to the user.

* * * * *